Solving System Problems with Machine Learning

Ion STOICA

Computer Science Division, University of California, Berkeley, 465 Soda Hall, CA 94720-1776, USA

istoica@cs.berkeley.edu

Abstract: Over the past decade, Machine Learning (ML) has achieved tremendous successes and has seen wide-scale adoption for human-facing tasks, such as visual recognition, speech recognition, language translation and medical diagnosis. However, going forward, we contend that ML has an even higher potential for impact by solving hard systems problems, such as improving industrial processes, supply chain optimization, and automatic program generation. One challenge is that solving many of these problems require solutions that are provably correct, which is at odds with the ML techniques which are stochastic in nature. In this paper, we consider this challenge and propose two approaches of using ML to solve system problems. The first approach, called *correct by construction*, is to generate provably correct solutions, for example, by starting from a correct solution, and applying ML-guided transformations that preserve the solution's correctness. The second approach, called *trust but verify*, is to generate solutions whose correctness can be (efficiently) verified, and then keep generating solutions until we find a correct one. To illustrate these approaches, we present several examples in the area of software systems, and show how using ML can provide significant improvements over state-of-the-art solutions which were refined over decades.

Keywords: Machine Learning, Systems, Reinforcement Learning, Optimization, Data structures, SQL.

1. Introduction

Machine Learning (ML) is beginning to have the impact on human society that has long been anticipated. In the past decade, we have seen ML move from labs to production with the wide-scale adoption of human-facing technologies in visual recognition, speech recognition, language translation and medical diagnosis.

Many of these breakthroughs have been enabled by large-scale computer systems capable of processing massive amounts of data to learn patterns and make predictions. The success of big data and big computation are driving organizations to process even more data to unlock even more value from this data. In most cases, the value is realized from the decisions (or actions) the data enables, such as deciding what content to display, what medical tests or treatment to recommend, how to respond to a voice command, and how to flag anomalies.

At the same time, ML has started being used to build better systems by optimizing their architecture and performance. Examples are designing better neural network architectures, improved scheduling, and video delivery.

This positive feedback loop of systems improving ML, and ML improving systems is an exciting development that will accelerate the progress of both systems and ML.

1.1 Human tasks

Much of the recent progress in ML has been on "human-tasks". By "human-tasks", we mean cognitive and recognition tasks, including image and speech recognition, language translation, and playing games, such as computer games, chess, and go. The recent progress on solving these â€œhuman-levelâ€ tasks has been nothing short of extraordinary, resulting in solutions that have matched or even outperformed humans on these tasks [33, 38, 39].

1.2 Systems (non-human) tasks

While solving human level tasks has driven the recent advancements in ML, going forward, we believe that ML will have an even bigger impact on the economy and our society by solving non-human or systems tasks. These tasks include improving industrial processes, optimizing architectures and system performance, and synthesize programs.

However, using ML to solve non-human tasks poses new challenges. While a solution solving a human task does not need to be provably correct as long as it matches or exceeds the human accuracy, this is not the case for many non-human tasks which need to provide provably correct solutions. Examples of such tasks are controlling industrial processes, providing the result of a database query, or the output of a program.

The fundamental challenge of applying ML to system problems is thus the mismatch between the stochastic nature of most ML techniques, and the need for provable correctness guarantees. To address this challenge, we need to reframe

https://doi.org/10.24846/v28i2y201901 ICI Bucharest © Copyright 2012-2019. All rights reserved

the problem. In this paper, we consider two approaches that allows us to reframe a system problem to leverage ML techniques without compromising the solution's correctness: *Correct by Construction*, and *Trust but Verify*. Table 1 summarizes these approaches.

Thus, the key differences between the two approaches is that the former ensures that every intermediate solution is correct, while the later approach only guarantees that the final solution is correct.

Table 1. Approaches of applying ML to non-human tasks to generate provably correct solutions.

Approach	Description	Example
Correct by Construction	**(a)** Start from a correct solution that preserve the correctness in order to improve the solution along some dimension, such as performance **(b)** Start from a solution (not neecessarily correct) and iteratively apply transformations to build a provably correct solution	**(a)** Packet classifier (see Sec. 3.1). **(b)** Join optimization (see Sec. 3.4).
Trust but Verify	Continuosly generate and verify solutions until a correct solution is found	Program synthesis (see Sec. 4.1); learned indexes (see Sec 4.2)

1.2.1 Correct by Construction

We consider two approaches to generate provable correct solutions using ML.

The first is to start with a solution that is correct and then apply a sequence of transformations, selected by an ML model, so that each transformation guarantees that the correctness and the semantics of the solution are preserved. The problem is then to come up with a sequence of such transformations and the order in which to apply these transformations, such that to improve the solution along some dimension, such as reducing the computation complexity or the space complexity. An example is optimizing the execution of an SQL query. In this case, we can start from an unoptimized but correct query plan, Q. Then, we optimize Q by applying one or more transformations, such as join reordering or pushing down predicates, to reduce its cost [6]. The key point here is that none of these transformations affect the correctness of Q.

The second approach is to build the solution iteratively, such that we are guaranteed that the final result, but not necessary the intermediate ones, is correct. One example is deciding in which order to join the tables in the set. Because the join operation is both commutative and associative, we are guaranteed that once we joined all tables the result is correct. Obviously, as long as we have not joined all tables the result is not guaranteed to be correct, as we might miss data.

1.2.2 Trust but Verify

In this case, we use an ML model to generate solutions until we find at least a correct one. This approach assumes that it is possible to (efficiently) verify a solution. One example is synthesizing a program from input-output examples. Upon synthesizing a program, we can check that it computes the desired output given the corresponding input.

A variant of this approach is to verify the prediction of an ML prediction.

2. Background: Machine Learning Techniques

In this section, we briefly present the two basic techniques used to solve the systems tasks we are considering in this paper: supervised learning and reinforcement learning.

2.1 Supervised Learning

Supervised learning is one of the most common and most successful techniques used in machine learning. Supervised learning is at the core of the recent advancements in ML, in particular, in solving human-level tasks such as image recognition, speech recognition, and language translation.

Fundamentally, supervised learning learns a function approximation, $f()$, from a set of input items, X, to a set of labels, L. For example, in the case of image recognition, the X consists of a set of images, and L consists of all labels or categories associated with images in X. Supervised learning

has two distinct phases: *training* and *prediction*. During training, we learn function *f*() (also called model) from a *labeled* dataset consisting of input/output examples, e.g., Imagenet [21]. During prediction, *f*() takes an unlabeled input, *a*, and generates its label, *f*(*a*). During the past decade, deep neural networks (DNNs) have revolutionized supervised learning with their ability to learn complex functions on unstructured inputs.

2.2 Reinforcement Learning

With reinforcement learning (RL) [43] , a software agent continuously interacts with the environment by taking actions. Each action can change the state of the environment and generate a "reward". The goal of RL is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward. An RL algorithm that uses a DNN to approximate the policy is referred to as a Deep RL algorithm. Figure 1 shows the components of the reinforcement learning system.

Recently, RL has shown the ability to solve some of the most difficult problems to date. These include learning to play Atari video games better than humans [33] and defeating the Go world champion [38]. In the case of playing Atari games, the environment is the game engine, the action is clicking on a particular key of the controller or keyboard, the observation is a screen shot, and the reward the score (shown on the screen). Similarly, in the case of Go, the environment is the board, the state is the position of the pieces on the board, the action is moving a piece on the board, and the reward is the results of the game.

As we will see next, due to its iterative nature and power, RL also plays an important role for solving non-human tasks.

3. Correct by Construction

There are two approaches to build correctly provable solutions using ML.

The first is to start with a correct solution and apply a sequence of transformations, where each transformation preserves the correctness of the solution. The goal is to chose a sequence of transformations such that to improve the solution along some dimension, like performance. Note that this approach trivially guarantees that at every step the solution is correct.

The second approach is to start from a solution (not necessary correct or complete) and apply a sequence of given transformations such we arrive to a provable correct solution. An example is finding the order in which we execute a set of operations, such that to optimize the performance. If the operations are both commutative and associative, any execution order will provide the same result.

In the reminder of this section, we provide four examples. The first tree illustrate the first approach and they are: building an efficient decision tree for packet classification, learning a order in which to apply the optimization phases to a program so we improve its execution speed, and optimizing the bit rate of a video streamed over the internet. The last example, which optimizes the join ordering of a SQL query, illustrates the second approach.

3.1 NeuroCuts: Network Packet Classification

3.1.1 The Problem

Packet classification is a building block for many network functionalities, including firewalls, access control, traffic engineering, and network measurement [11,25,45]. The goal of packet

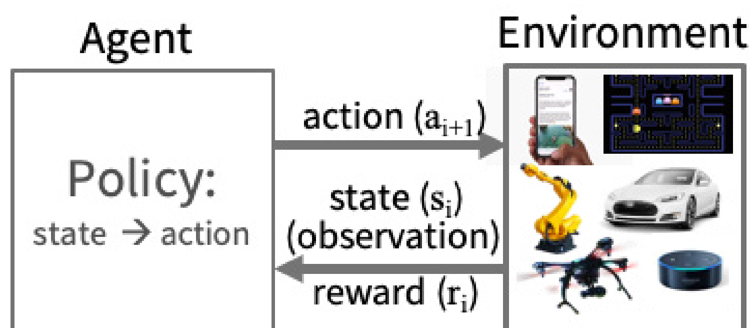

Figure 1. Reinforcement learning (RL) systems. An agent interacts with the environment by taking actions that can modify the state of the environment to learn a policy that maximizes a reward.

ICI Bucharest © Copyright 2012-2019. All rights reserved

classification is to match a given packet to a rule from a set of rules, which dictates how a router processes that packet, e.g., forward it with high priority, drop it, or forward it to a specific network device. Given its importance, packet classification has received considerable attention over the past *20 years*, with many solutions being proposed.

Figure 2 illustrates the typical architecture of the packet classifier. The packet classifier consists of a table containing a set of rules. A rule is a mapping between a filter on the fialds in the packet header and an action specifying how to act on the packets matching that filter. A filter consists of (a) two prefixes for both the source and destionatio IP addresses, (b) two ranges for both source and destination port numbers, and (c) the protocol type (e.g., TCP, UDP). A packet matches a filter if packet's destination/source IP address matches the the filter's destination/source prefixes, the destination/source port number is contained in the filter's destination/source port number ranges, and the packet's protocol type matches the filter's protocol type. Since filters can overlap, the rules entries in the table also contain a priority field that specifies which rule should be applied when the packet matches multiple rules.

Ideally, a packet classifier should have both (a) low computation complexity, as we need to classify packets at the line rate, and (b) low space complexity, as high-speed memory is notoriously expensive and power hungry. Unfortunately, packet classification is similar to the point location problem in a multi-dimensional geometric space, which exhibits a hard tradeoff between computation and space complexities: the fields in the packet header represent the dimensions in the geometric space, a packet is represented as a point in this space, and a rule as a hypercube. In a d-dimensional geometric space with n non-overlapping hypercubes and d dimensions, this problem has either (i) a lower bound of $O(\log n)$ time and space, or (ii) a lower bound of time and $O(n)$ space [12]. Worse yet, packet classification is a harder problem since, as mentioned above, it allows rules to overlap.

Existing solutions for packet classification can be divided into two broad categories. Solutions in the first category are hardware-based. They leverage Ternary Content-Addressable Memories (TCAMs) to store all rules in an associative memory, and then match a packet to all these rules in parallel [22]. As a result, TCAMs provide constant classification time, but come with significant limitations. TCAMs are inherently complex, and this complexity leads to high cost and power consumption. This makes TCAM-based solutions prohibitive for implementing large classifiers [45].

The solutions in the second category are software based. These solutions build sophisticated in-memory data structuresâ ”typically decision treesâ" to efficiently perform packet classification [25]. While these solutions are far more scalable than TCAM-based solutions, they are slower, as the classification operation needs to traverse the decision tree from the root to the matching leaf. Most existing solutions for packet classification aim to build a decision tree that exhibits low classification time (i.e., time complexity) and memory footprint (i.e., space complexity) [45]. Given a decision tree, classifying a packet reduces to walk the tree from the root to a leaf, and then chose the highest priority rule associated with that leaf. These solutions employ two general techniques to build decision trees for packet classification:

- *Node cutting:* split nodes in the decision tree by "cutting" them along one or more dimensions.

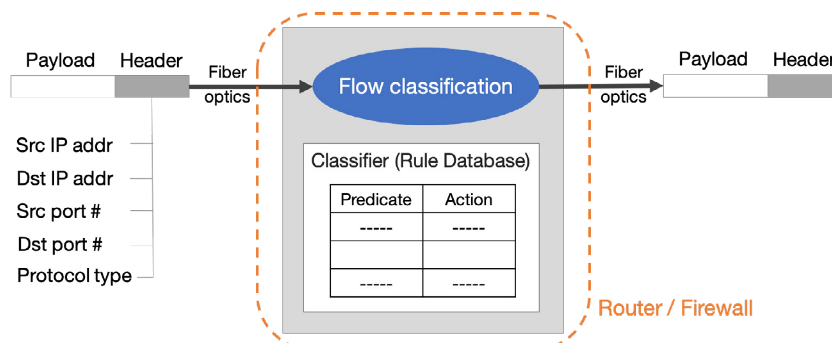

Figure 2. The implementation of a packet classifier. Upon the arriving of a packet, the packet is matched to a rule in the table, and the corresponding action in the rule is applied on the packet. The rule table can contain 100K rules, or more.

- *Rule partition:* if a rule has a large size along one dimension, cutting along that dimension will result in that rule being added to many nodes.

Figure 3 shows a simple example of a decision tree for six rules in a space with two options.

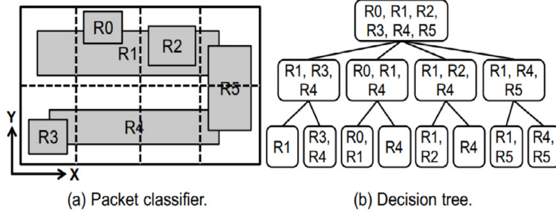

Figure 3. An example of six rules in a two dimensional space, and a possible decision tree

3.1.2 RL Formulation

There are two possible approaches to solve the classification problem using ML. The first is to use supervised learning to directly perform packet classification. In this case, we just learn a model whose input consists of the fields in a packet header, and the output is the class to which the packet belongs to. Unfortunately, this solution is not good enough because it cannot guarantee 100% accuracy. Given the fact that one of the main use cases of packet classification is security, having a packet classified incorrectly is a non starter.

Instead, one can use RL to build a decision tree that is provably correct [27]. Figure 4 shows the process of building the decision tree formulated as an RL problem. In short, we define the state, action, and reward, as follows:

- *state*: The current decision tree. The initial state consists of a single node that covers all the rules. Upon a packet arrival, we search for the leaf in the decision tree that â€œcontainsâ€ that packet, and then we linearly search across all rules belonging to that leaf to find the rule matching the packet.

- *action*: Pick a leaf node and either split it along a dimension or duplicate it. This basically reduces the number of rules associated with a node, which in turn reduces the classification complexity.

- *reward*: Since we want to achieve both low computation and space complexities, the reward is the negative of a linear combination between the tree depth (which determine the computation complexity) and the tree size (which determines the space complexity.

3.1.3 Results

Neurocuts outperforms the state-of-the-art solutions [27]. It improves the median of the classification time by 18% compared to existing solutions, and improves any existing solution by 3×, either in terms of memory capacity or classification time.

3.2 Autophase: Program optimization

3.2.1 The Problem

High-Level Synthesis (HLS) automates the process of creating digital hardware circuits from algorithms written in high-level languages. Modern HLS tools [7, 15, 46] use the same frontend as the traditional software compilers. They rely on traditional compiler techniques to optimize the input program's intermediate representation (IR) and produce circuits in the form of register transfer level (RTL) code. Thus, the quality of compiler front-end optimizations directly impacts the performance of HLS-generated circuit.

Program optimization is a notoriously difficult task. A program must be just in "the right form" for a compiler to recognize the optimization opportunities. This is a task a programmer might

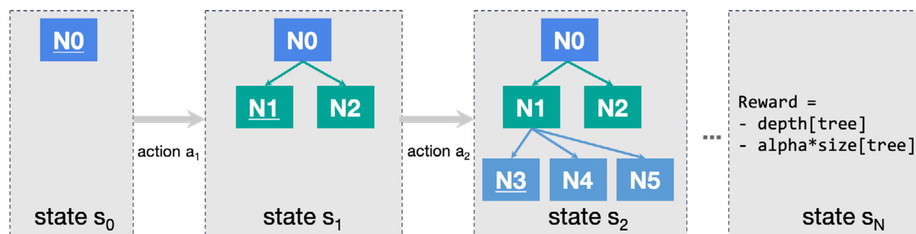

Figure 4. An illustration of applying RL to building a decision tree for packeet classification. The state represents the current decision tree, the action represents which leaf node to split or duplicate, and the reward is a negative of a linear combination between the tree depth and size.

ICI Bucharest © Copyright 2012-2019. All rights reserved

be able to perform easily, but is often difficult for a compiler. Despite a decade of research on developing sophisticated optimization algorithms, an expert designer can still produce RTL that outperforms the results of HLS.

The optimization of an HLS program consists of applying a sequence of analysis and optimization phases, where each phase in this sequence consumes the output of the previous phase, and generates a modified version of the program for the next phase. Unfortunately, these phases are not commutative which makes the order in which these phases are applied critical to the performance of the output. It is easy to construct programs in which the order in which the optimization phases are applied can be the difference between the program running in $O(n^2)$ versus $O(n)$ [3].

It is thus crucial to determine the optimal phase ordering to maximize the circuit speeds. Unfortunately, not only is this a difficult task, but the optimal phase ordering may vary from program to program. Furthermore, it turns out that finding the optimal sequence of optimization phases is an NP-hard problem, and exhaustively evaluating all possible sequences is infeasible in practice. Again, like in the case of the previous example.

3.2.2 RL Formulation

In this problem, we assume we want to apply n passes out of a total of $k \geq n$ pasess, which give us a search space of size [3]. Given a set of program features and the history of the passes already applied, the goal of RL is to learn the next best optimization pass a to apply so that to minimize the overall cycle count of the generated hardware circuit. Examples of program features are the number of memory accesses, number of returning instruction, etc. We formulate this problem as an RL problem as follows:

- *State / observation:* The program features and the sequence of optimization phases that have been applied so far on the program.
- *Action:* The next optimization phase to be applied.
- *Reward:* The difference between (1) the cycle count of the previous configuration, and (2) the cycle count of the current configuration. Note that positive reward corresponds to a reduction in the cycle count, and hence an increase in performance.

3.2.3 Results

Autophase improves the performance of the best configured compilers by 29%s, and matches other state-of-the-art solutions, while requiring much fewer samples [3]. More importantly, unlike existing state-of-the-art solutions, our reinforcement learning solution can generalize to more than 12,000 different programs after training on as few as a hundred programs for less than ten minutes.

3.3 Pensieve: Adaptive Video Streaming

3.3.1 The Problem

HTTP-based adaptive streaming (e.g., DASH [2]) is the predominant form of video delivery today. By transmitting video using HTTP, content providers are able to leverage existing CDN infrastructure and maintain simplified (stateless) backends. Further, HTTP is compatible with many client-side applications, including web browsers and mobile applications.

In DASH systems, videos are stored on servers as multiple chunks, each of which represents a few seconds of the overall video. Each chunk is encoded at several discrete bitrates, where a higher bitrate corresponds to a higher quality and thus a larger chunk size. Chunks across bitrates are aligned to support seamless quality transitions, i.e., a video player can switch to a different bitrate at any chunk boundary without fetching redundant bits or skipping parts of the video.

Adaptive bitrate (ABR) is the primary technique to optimize video quality. The algorithms based on this technique run on client-side video players and dynamically choose a bitrate for each video chunk (e.g., 5-second block). ABR algorithms make bitrate decisions based on various observations such as the estimated network throughput and playback buffer occupancy. Their goal is to maximize the userâ€™s quality of experience (QoE) by adapting the video bitrate to the underlying network conditions. However, selecting the right bitrate is challenging due to (1) the variability of network throughput [13, 42, 50, 51], (2) the conflicting video QoE requirements (e.g., high bitrate, minimal rebuffering,

smoothness), (3) the cascading effects of bitrate decisions (e.g., selecting a high bitrate may drain the playback buffer to a dangerous level and cause rebuffering in the future), and (4) the coarse-grained nature of ABR decisions.

The majority of existing ABR algorithms are using a set of simple heuristics for making bitrate decisions based on estimated network throughput [17, 42] playback buffer size [14, 41], or a combination of the two signals [26]. These schemes require significant tuning and do not generalize to different network conditions and QoE objectives. The state-of-the-art approach, MPC [49], makes bitrate decisions by solving a QoE maximization problem over a horizon of several future chunks. By optimizing directly for the desired QoE objective, MPC can perform better than approaches that use fixed heuristics. However, MPC's performance relies on an accurate model of the system dynamics, including the predicted network throughput. As a result, MPC is brittle in the face of even slightly inaccuracies of the performance model.

3.3.2 RL Formulation

It turns out that it is quite simple to formulate the problem of optimizing the adaptive bit rate switching as an RL problem [29]:

- *Status:* The buffer occupancy and the bitrate of the stream, as well as the available bandwidth measurements.
- *Action:* Decide the bitrate of the next chunck.
- *Reward:* The quality as perceived by the user, i.e., QoE.

3.3.3 Results

Using a broad set of network conditions and reward metrics, [29] shows that Pensieve matches or exceeds the best existing schemes, by imporving the QoE by up to 25%. Furthermore, Pensieve has the ability to generalize to unseen network conditions and video properties.

3.4 DQ: Join Optimization in Databases

3.4.1 The Problem

Arguably, the most expensive and difficult operation in databases is the *join* operation. It is expensive because the join of two tables can generate a result that is as large as the carthesian product between the two tables. It is difficult because it is very hard to predict the size of the result without doing the join itself, and this size can be anywhere between zero and the size of chartesian product of the two tables.

The cost of joining multiple tables depends on the order in which these tables are joined. Figure 5 shows a simple example of joinning three tables *A*, *B*, and *C*, respectively. In this example, we trivially assume the cost of a join plan is the total number of rows across all tables it generates. For example, the join plan shown in Figure 5(a) first joins *A* and *B* to generate a table with 7 rows, and then joins

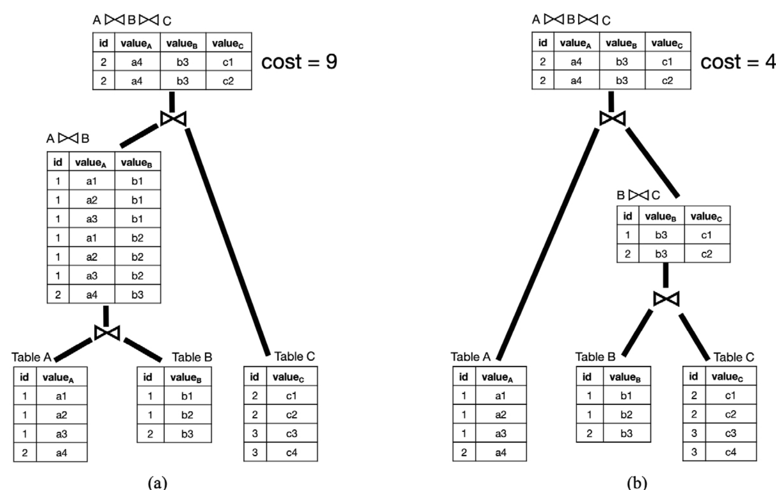

Figure 5. A simple example of joining three tables *A*, *B*, and *C*, respectively. There are three possible ways to join these tables. Here we show two: (a) $(A \bowtie B) \bowtie C$ with a cost of 9, and (b) $A \bowtie (B \bowtie C)$ with a cost of 5. Here the cost is simply computed as the total number of rows in all.

ICI Bucharest © Copyright 2012-2019. All rights reserved

this result table with table C to generate the final result, which consists of 2 rows. Thus, the total cost is 9. In contrast, if we join first B and C, as shown in Figure 5(b), then we get a cost of only 4. Note, that there is another possible join plan where we first join A and C, not shown in Figure 5.

Given n tables, our goal is to find a query plan, i.e., an order in which we join the tables, that minimizes the cost of joining all n tables. Since the join operation is both commutative and associative, the order in which we join the tables has no impact on the result. As such, to guarantee the result's correctness we just need to guarantee that all n tables are joined exactly ones.

Unfortunately, there are $n!$ possible ways in which we can join n tables. Even for modest values of n, evaluating all possible join orders is not feasible. Given its difficulty and performance implications, join optimization has been studied for more than *four decades* [37] and continues to be an active area of research [30, 34, 37]. To reduce the problem's combinatorial complexity, solutions typically employ *heuristics*. For example, classical System R-style dynamic programs often restrict its search space to certain shapes (e.g., "left-deep" join plans). Since for large joins these heuristics are not always enough to reduce the search space to a tractable size, query optimizers sometimes apply further heuristics, such as genetic [1] or randomized [34] algorithms. Unfortunately, in edge cases, these heuristics are brittle and can generate poor plans [24].

3.4.2 ML Formulation

In the light of the recent advances in ML, a new trend in database research explores replacing programmed heuristics with learned ones [4, 18, 19, 28, 30–32, 35].

Next, we present one of the most recent works in this line of research that leverages RL to solve the optimization problem, called *Deep Queries (DQ)* [20]. The join ordering problem is formulated as an RL problem as follows:

- *state:* The set of tables joined so far.
- *action:* The table to join next.
- *reward:* The negative of the estimated cost. We take the negative since the RL algorithms typically aim to maximize the reward, while in this case we want to minimize the cost.

Figure 6 illustrates the application of RL to the join optimization problem. The entire state is represented as a binary string with the length equal to the number of tables we want to join. An 1 bit means that the corresponding table has been already joined, while a 0 bit means that the table has not been joined yet. For the initial state, we randomly select two tables to join, in this case tables $T0$ and $T4$. Then, at each step, the RL algorithm selects the next table to join. In our example, the RL algorithm first selects $T8$, then $T1$, and so on until all tables are joined. As a policiy, the solution uses DQN, an approximation of the classic Q-learning algorithm by a neural network [33].

With any RL algorithm, one natural question is sample efficiency, i.e., how many query examples we need to learn a good policy. The RL algorithms are notoriously data-inefficient. Indeed, typical RL settings, such as the Atari games [33], require hundreds of thousands of training examples! Fortunately, in our case, we can exploit the optimal subplan structure specific to join optimization to collect a large amount of high-quality training data. From a single query that passes through a native optimizer, not only are the final plan and its total cost collected as a training example, but all of its subplans. For instance, planning an 18-relation join query (i.e., Query 64 in the

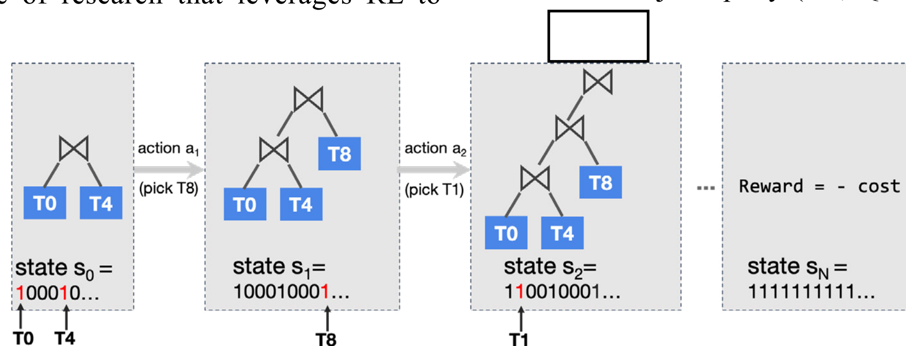

Figure 6. An illustration of applying RL to optimize joins. The state is represented by a binary string indicating which tables have been already joined, the action indicates which table should be joined next, and the reward is the negative of the cost.

TPC-DS benchmark) using the traditional bushy optimzier can yield up to 600,000 training data points. This is because, in addition to the final results, we can use every partial result in the query plan as a training example. For instance, when we execute the query plan in Figure 5, we learn not only the cost of generating the result, but also the cost of the sub-plane joining *A* and *B*.

3.4.3 Results

While simple, this solutions achieves remarkable results [20]. Under a variety of cost models, DQ achieves speedups in planning times (up to >200×) relative to dynamic programming enumeration, while essentially matching the execution times of optimal plans computed by the native enumeration-based optimizers. Even more impressivley, DQ is particularly robust under non-linear cost models such as memory limits or materialization. On two simulated cost models with significant non-linearities, DQ improves on the plan quality of the next best heuristic over a set of DQ baselines by 1.7× and 3×, respectively. In summary, DQ approaches the optimization time efficiency of programmed heuristics *and* the plan quality of optimal enumeration.

4. Trust but Verify

The main assumption behind this approach is that it is possible to (efficiently) verify a solution's correctness. There are two patterns in which to apply this approach.

The first pattern is during the training phase. We continuously generate solutions and verify their correctness until at least a correct solution is found.

The second pattern is during the prediction phase. We verify each prediction, and, if incorrect, we possibly imvoke a provably correct but less desirable (i.e., slow) one.

4.1 Autopandas: Program Synthesis

This example illustrates the first pattern of our approach. We generate solutions (i.e., programs) until we find a correct one. Here the "correctness" is defined by checking whether the program generates a set of expected outputs, given a set of inputs.

4.1.1 The Problem

Developers are increasingly using powerful APIs, often packaged in popular libraries, to build sophisticated applications. Using such APIs allows developers to (1) build applications faster as they obviate the need of writing large amounts of code, and (2) build more robust applications, as the code behind these APIs is typically well tested.

Unfortunately, the price to pay for these powerful and versatile APIs is a steep learning curve, as often there are hundreds of APIs one needs to master for each type of application or workload. Indeed, popular Python libraries, such as NumPy and Pandas have each hundreds of APIs, and each of these APIs can have tens of arguments. Furthermore, the documentation of all of these APIs is of varying quality. Worse yet, modern APIs are frequently updated, so tutorials, blog posts, and other external resources on the API can quickly fall out of date. All these factors make it difficult for developers to learn the API sufficiently well to use it efficiently.

As a result, developers often resort to asking their more experienced colleagues or leverage on-line forums, such as StackOverflow. Unfortunately, none of these venues is ideal. It's not always that a developer has access to an expert when she needs one, and questions on StackOverflow might take days to answer.

The one alternative is to design systems to automatically answer these questions based on the hints provided by the users. One example of such hints are examples of inputs and outputs. Indeed, given an input, the developer often knows what is the output she wants, but doesn't know which function calls to use to get that output.

Thus, the problem we want to address is: *generate a sequence of API calls which applied on the given input(s) will generate the specified output(s)*. This is similar to the program synthesis problem, a notoriously difficult problem, which has received considerable attention over the past two decades. However, so far there has been little progress beyond simple program examples, typically written in domain specific languages (DSLs). Examples of past solutions include string processing [10, 36]. data wrangling [8, 9, 23], data processing [40, 47], database queries [48] and bit-vector manipulations [16].

ICI Bucharest © Copyright 2012-2019. All rights reserved

4.1.2 ML Formulation

Recently, several works have aimed to synthesis small programs for several popular APIs and languages. Of these, here we consider Autopandas [5] which uses ML to generate short sequence of APIs calls from input/output examples for the popular Pandas library, the de facto standard for data scientists.

The approach taken in [5] is to use a neural network model that learns to "predict" (generate) programs from one or more pairs of input/output examples. For each program being generated, we then check whether the output it creates matches the desired output, and, if yes, we select that program. If not, we continue to generate new programs until we find one that computes the desired output from the given input. Figure 7 illustrate this approach. A variant of this approach is not to stop when we find the first program generated the desired output, but continue until a timeout expires. The main reason is that the problem is in most cases unspecified, i.e., given an input, there are many programs computing the same output. This gives a chance to the developers to pick one of the many possible valid programs.

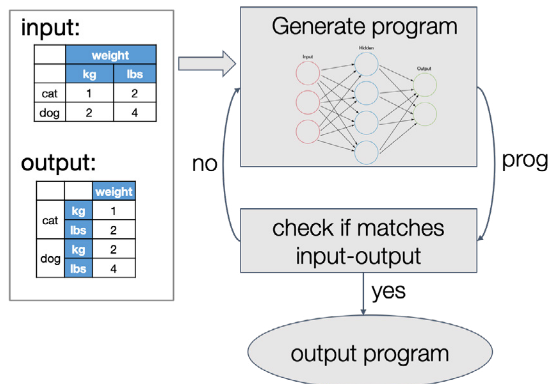

Figure 7. Using a model to generate programs until a correct one is found. A correct program is defined as the one generating the expected output from a given input.

The main challenge with this approach is that the state-space is huge. For example, even if we consider single-function programs, the number of such programs is . In practice, this huge search space makes it very hard, if not infeasible, to train an accurate model to solve this problem.

Unfortunately, there is a key observation that allows us to dramatically reduce the search space: API functions often place constraints on the arguments beyond type. In Python, they can also accept multiple types for a single argument, with different constraints for each type. Thus, many combinations of arguments do not make sense.

Autopandas leverages this information and combines expert knowledge with stoochastic search to address this problem. In particular, for each API function, Autopandas employs a generator, written by an expert that captures the semantics constraints of the function's arguments. Then, for each argument of each function it trains a neural network to predict the arguments of that function. This considerably reduces the search space. For single function programs, Autopandas reduces the search space from to a reduction!

4.1.3 Results

Using real-world banchmarks from Pandas tutorials and StackOverflow questions, Autopandas can efficiently provide the same answer as the one provided by human experts in 65% of cases for single-function programs. For two-function programs, these numbers are 40%, and 73%, respectively. And, for three-function programs, these numbers are 35%, and 60%, respectively. For context, 95% of the answers in StackOverflow are no longer than three functions.

While we are still far from solving the general synthesis problem, these are promising results, especially as it targets a much broader domain than what current state-of-the-art programming-by-example synthesis systems handle.

4.2 Learned Index Structures

This example illustrates the second pattern of the "trust but verify" approach, i.e., verify every prediction, and, if incorrect, fall back on a provably correct but less efficient one.

4.2.1 The problem

To improve data access performance in many applications, such as databases and file systems, designers use sophisticated data structures such as B-Trees, hash tables, and Bloom filters. Because of their importance, indexes have been some of the most studied and optimized data structures over the past decades. However, despite the attention they received, the existing solutions assume nothing about data distribution, and, as a result, they fail to take advantage of common access patterns and distributions.

4.2.2 The ML formulation

Recent work [19] has proposed to fill this gap by using ML to learn indexes for a given data distribution and access pattern. The intuition behind this approach is that fundamentally an index maps a key to the location of the record associated to that key. This is essentially a prediction operation: given a key, "predict" the location of the record with that key. An ML approach can learn a model for this mapping by training the model against queries to the dataset. By doing so, the model will be optimized for the query workload and key distribution in the dataset.

We assume a setting in which data consists of (key, value) pairs, also called records, sorted by their keys. These records are stored in pages (or blocks), *n* records per page. A page reperesents the unit of storage access, i.e., the reads and writes occur at the page granularity. Thus, a range-based index, such as B-Trees, needs to predict the page containing the record corresponding to that key. To perform this prediction, [19] proposes using neural networks to learn the mapping from key to the page containing that key.

An evolution of this approach is to use a tree of models, instead of a single model, where each model in the tree selects one of its descendents. Eventually, the leaf models point to pages. Using a tree of models, enables us to specialize models for different regions of the key space, which is highly desirable when the density of the keys in the key space vary widely.

Unfortunately, using a neural network model to predict the location of a record is not 100% accurate. This means that sometimes, we will get a wrong page, i.e., a page that does not contain the given key. The important point to note here is that when this happens it will only affect the performance and not correctness. This is because we can easily verify whether the answer is correct. Since ultimately the record we are looking stores the key, we can always check that we got the right record. If the model predicts the wrong page, we can use a traditional index structure (or even scan the data) to locate the desired record. While locating the record when the prediction fails can take significantly longer, the hope is that the prediction operation takes much less than using a traditional index structure, and that the model accuracy is high enough.

4.2.3 Result

The initial results in [19] show that by using neural nets we are able to outperform cache-optimized B-Trees by up to 70% in speed while saving an order-of-magnitude in memory over several real-world data sets.

5. Conclusions

In this paper, we argue that while the ML has achieved tremendous successes over the past decade by solving many human-level tasks, such as video recognition, speech recognition, and language translation, going forward, ML has the potential to have an even bigger impact on solving hard systems problems by optimizing existing algorithms that either exhibit combinatorial complexity or employ brittle heuristics.

However, solving these problems require solutions that are provably correct, which is difficult to achieve by using an end-to-end ML approach, as ML techniques, such as deep neural networks (DNNs), are stochastic in nature. To address this challenge, in this paper we discussed two approaches to solve these problems.

The first approach is to generate provably correct solutions. There are two ways to do this: (a) start from a correct solution and apply transformations that preserve the solution's correctness, and (2) chose the starting solution and the transformations such that to guarantee we end up with a correct solution.

The second approach is to generate solutions whose correctness can be (efficiently) verified. In this case, we keep generating solutions until we find a correct one.

To illustrate these approaches, we presented several examples in the area of software systems. While the results are promising, we believe this is just the beginning, and we will see a rapid progress in applying ML to virtually any system and engineering problem for which the current solutions fall short. We hope this paper will inspire the reader to apply these approaches, or come up with new ones, to her own area of research.

ICI Bucharest © Copyright 2012-2019. All rights reserved

REFERENCES

1. [n. d.]. *PostgreSQL: Genetic Query Optimizer*. <https://www.postgresql.org/docs/11/static/geqo.html>.

2. Akamai (2016). *dash.js*. <https://github.com/Dash-Industry-Forum/dash.js/>.

3. Ameer Haj Ali, Qijing Huang, William Moses, John Xiang, Ion Stoica, Krste Asanovic & John Wawrzynek (2019). AutoPhase: Compiler Phase-Ordering for High Level Synthesis with Deep Reinforcement Learning*, CoRR abs/1901.04615* <http://arxiv.org/abs/1901.04615>.

4. Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, & Matei Zaharia (2017). *Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes*. <https://dawn.cs.stanford.edu/2018/01/11/index-baselines/>.

5. Rohan Bavishi, Caroline Lemieux, Neel Kant, Roy Fox, Koushik Sen & Ion Stoica (2018). Neural Inference of API Functions from Input-Output Examples, *Workshop on ML for Systems at NeurIPS*.

6. Nicolas Bruno, YongChul Kwon & Ming-Chuan Wu (2014). Advanced Join Strategies for Large-scale Distributed Computation. In *Proc. VLDB Endow. 7(13)* (pp. 1484–1495). 2150-8097 <https://doi.org/10.14778/2733004.2733020>

7. Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown & Jason H Anderson (2013). LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems, *ACM Transactions on Embedded Computing Systems (TECS)*, *13*(2), 24.

8. Yu Feng, Ruben Martins, Osbert Bastani, & Isil Dillig (2018). Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)* (pp. 420–435). ACM, New York, NY, USA 978-1-4503-5698-5 <https://doi.org/10.1145/3192366.3192382>

9. Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig & Swarat Chaudhuri (2017). Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples, *SIGPLAN Not. 52*(6), 422–436. 0362-1340, <https://doi.org/10.1145/3140587.3062351>.

10. Sumit Gulwani (2011). Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)* (pp. 317–330). ACM, New York, NY, USA. 978-1-4503-0490-0, <https://doi.org/10.1145/1926385.1926423>.

11. Pankaj Gupta & Nick McKeown (1999). Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*.

12. Pankaj Gupta & Nick McKeown (2001). *Algorithms for packet classification*.

13. Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown & Ramesh Johari (2012). *Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard*.

14. [Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell & Mark Watson (2014). A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)* (pp. 187–198). ACM, New York, NY, USA, 978-1-4503-2836-4, <https://doi.org/10.1145/2619239.2626296>.

15. Intel. [n. d.]. *Intel FPGA SDK for OpenCL*. <https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/developer-zone.html>.

16. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010). Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)* (pp. 215–224). ACM, New York, NY, USA, 978-1-60558-719-6 <https://doi.org/10.1145/1806799.1806833>.

17. J. Jiang, V. Sekar & H. Zhang (2012). *Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE*.

18. Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz & Alfons Kemper (2018). Learned Cardinalities: Estimating Correlated Joins with Deep Learning, *arXiv preprint arXiv:1809.00677*.

19. Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean & Neoklis Polyzotis (2018). The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)* (pp. 489–504). New York, NY, USA, 978-1-4503-4703-7.

20. Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein & Ion Stoica (2018). Learning to Optimize Join QueriesWith Deep Reinforcement Learning. *CoRR abs/1808.03196* <http://arxiv.org/abs/1808.03196>.

21. Alex Krizhevsky, Ilya Sutskever & Geoffrey E Hinton (2012). ImageNet Classification with Deep Convolutional Neural Networks, *NIPS,* 1106-1114.

22. Karthik Lakshminarayanan, Anand Rangarajan, & Srinivasan Venkatachary (2005). Algorithms for advanced packet classification with ternary CAMs. In *SIGCOMM CCR*.

23. Le14 Vu Le & Sumit Gulwani (2014). FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)* (pp. 542–553). ACM, New York, NY, USA, 978-1-4503-2784-8, <https://doi.org/10.1145/2594291.2594333>.

24. Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper & Thomas Neumann (2015). How good are query optimizers, really? In *Proceedings of the VLDB Endowment 9*(3) (pp. 204–215).

25. Wenjun Li, Xianfeng Li, Hui Li & Gaogang Xie (2018). CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE INFOCOM*.

26. Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen & D. Oran (2014). Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale, *IEEE Journal on Selected Areas in Communications*, *32*(4), 719-733.

27. Eric Liang, Hang Zhu, Xin Jin & Ion Stoica (2019). Neural Packet Classification. *CoRR abs/1902.10319,* <http://arxiv.org/abs/1902.10319>.

28. Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo & 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* (pp. 631-645). ACM.

29. Hongzi Mao, Ravi Netravali & 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)* (pp. 197-210). New York, NY, USA.

30. Ryan Marcus & Olga Papaemmanouil (2018a). Deep reinforcement learning for join order enumeration, *arXiv preprint arXiv:1803.00055*.

31. Ryan Marcus & Olga Papaemmanouil (2018b). Towards a Hands-Free Query Optimizer through Deep Learning, *arXiv preprint arXiv:1809.10212*.

32. Michael Mitzenmacher (2018). A Model for Learned Bloom Filters and Related Structures, *arXiv preprint arXiv:1802.00884*.

33. 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 & Demis Hassabis. (2015). Human-level control through deep reinforcement learning, *Nature*, *518*(7540), 529-533. 0028-0836, <http://dx.doi.org/10.1038/nature14236>.

34. Thomas Neumann & Bernhard Radke (2018). Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 677-692). ACM.

35. Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke & S. Sathiya Keerthi (2018). Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings of the Second Workshop on Data Management for End-*

ICI Bucharest © Copyright 2012-2019. All rights reserved

To-End Machine Learning (DEEM'18), Article 4 (4 pages). ACM, New York, NY, USA, 978-1-4503-5828-6, <https://doi.org/10.1145/3209889.3209890>.

36. Emilio Parisotto, Abdelrahman Mohamed, Rishabh Singh, Lihong Li, Denny Zhou & Pushmeet Kohli (2017). Neuro-Symbolic Program Synthesis. In *ICLR 2017*. <https://www.microsoft.com/en-us/research/publication/neuro-symbolic-program-synthesis-2/>.

37. P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie & Thomas G Price (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (pp. 23-34). ACM.

38. 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), 484-489.

39. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan & Demis Hassabis (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, *NIPS*. arXiv:1712.01815, <http://arxiv.org/abs/1712.01815>.

40. Calvin Smith & Aws Albarghouthi (2016). MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)* (pp. 326–340). ACM, New York, NY, USA, 978-1-4503-4261-2, <https://doi.org/10.1145/2908080.2908102>.

41. Kevin Spiteri, Rahul Urgaonkar & Ramesh' K. Sitaraman (2016). *BOLA: Near-optimal bitrate adaptation for online videos*, 1-9. <https://doi.org/10.1109/INFOCOM.2016.7524428>.

42. Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu & Bruno Sinopoli (2016). *CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction*.

43. Richard S Sutton & Andrew G Barto (1998). *Introduction to reinforcement learning, Vol. 135*. MIT press Cambridge.

44. Immanuel Trummer & Christoph Koch (2017). Solving the Join Ordering Problem via Mixed Integer Linear Programming. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1025-1040). ACM.

45. Balajee Vamanan, Gwendolyn Voskuilen & T. N. Vijaykumar (2010). EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *ACM SIGCOMM*.

46. Xilinx (2015). *Vivado Design Suite User Guide - High-Level Synthesis*. <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug902-vivado-high-level-synthesis.pdf>.

47. Navid Yaghmazadeh, Xinyu Wang & Isil Dillig (2018). Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. In *Proc. VLDB Endow, 11*(5) (pp. 580-593). 2150-8097, <https://doi.org/10.1145/3187009.3177735>.

48. Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, & Thomas Dillig (2017). SQLizer: Query Synthesis from Natural Language. In *Proc. ACM Program. Lang. 1, OOPSLA*, Article 63 (26 pages). 2475-1421, <https://doi.org/10.1145/3133887>.

49. Xiaoqi Yin, Abhishek Jindal, Vyas Sekar & Bruno Sinopoli (2015). A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)* (pp. 325-338). New York, NY, USA, 978-1-4503-3542-3.

50. Yasir Zaki, Thomas Potsch, Jay Chen, Lakshminarayanan Subramanian & Carmelita Gorg (2015). *Adaptive congestion control for unpredictable cellular networks*.

51. X. K. Zou (2015). *Can Accurate Predictions Improve Video Streaming in Cellular Networks?*