Extending Relational Query Processing with ML Inference

Konstantinos Karanasos[1], Matteo Interlandi[1], Doris Xin[2*], Fotis Psallidas[1], Rathijit Sen[1], Kwanghyun Park[1], Ivan Popivanov[1], Supun Nakandal[3*], Subru Krishnan[1], Markus Weimer[1], Yuan Yu[1], Raghu Ramakrishnan[1], Carlo Curino[1]

[1]Microsoft [2]University of California, Berkeley [3]University of California, San Diego

ABSTRACT

The broadening adoption of machine learning in the enterprise is increasing the pressure for strict governance and cost-effective performance, in particular for the common and consequential steps of model storage and inference.

The RDBMS provides a natural starting point, given its mature infrastructure for fast data access and processing, along with support for enterprise features, such as encryption, auditing, and high-availability. To take advantage of all of the above, we need to address a key concern: *Can in-RDBMS scoring of ML models match (outperform?) the performance of dedicated frameworks?*

We answer the above positively by building *Raven*, a system that leverages native integration of ML runtimes (such as ONNX Runtime) deep within SQL Server and a unified intermediate representation (IR) to enable advanced cross-optimizations between ML and database operators. In this optimization space, we discover the most exciting research opportunities that combine DB/compiler/ML thinking. Our initial evaluation on real data demonstrates performance gains of up to 5.5× from the native integration of ML in SQL Server and up to 24× from cross-optimizations. An early preview of the ONNX Runtime integration is currently available with Azure's SQL Database Edge.

1. INTRODUCTION

Advances in machine learning (ML), first proven in high-value web applications, are fueling a trend towards digitally transforming almost every industry—in large part due to the excitement around using ML to complement traditional data analysis, discover new insights, and amplify weak signals.

However, safely and effectively adopting ML in enterprise settings comes with many new challenges across model training, tracking, deployment, and inference. We consider all those aspects of what we term *Enterprise Grade Machine Learning* as part of our broader research agenda [3], and focus this paper on model inference in particular.

As more and more data is analyzed and monetized, concerns about securing sensitive data and risks to individual privacy have been growing considerably [15]—this extends to ML models. In fact, based on interactions with enterprise customers, we expect that *storage and inference of ML models will be subject to the same scrutiny and performance requirements of sensitive/mission-critical operational data.*

When it comes to data, database management systems (DBMSs) have been the trusted repositories for the enterprise. They provide fast data access and processing, as well as a mature infrastructure that delivers features such as rich connectivity, transactions, versioning, security, auditing, high-availability, and application/tool integration. We thus propose to store and serve ML models from within the DBMS in order to extend the above described guarantees to models as well as data. However, given the current rudimentary support for ML in DBMSs, a key concern is to do so with no detriment to inference performance. This leads us to the key question we investigate in this paper: *Can in-RDBMS scoring of ML models match (outperform?) the performance of dedicated frameworks?*

In parallel, an interesting trend has emerged with respect to *inference* of ML models. Most widely studied or promising model families can be uniformly represented [25], and given a particular model, we can express how to score it on a given input using an appropriate algebra [30, 43]. These algebraic structures can then be executed on different environments and hardware [1, 11, 31, 36]. Among these efforts, ONNX is worth mentioning as a recent attempt for an open format to standardize ML model representation in an engine-agnostic manner, similar to the role of relational algebra in RDBMSs. Taken together, these observations suggest that we need to consider *how to incorporate ML scoring as a foundational extension of relational algebra and an integral part of SQL query optimizers and runtimes.*

Specifically, we are building Raven, a system that supports in-DB model inference and leverages sophisticated cross-optimizations and tight integration of ML runtimes in the DB to outperform common practical solutions by up to 24×.

In our vision, data scientists should be able to design and train ML models with their favorite ML framework. Once trained, these models, combined with any required data preprocessing steps and library dependencies, form what we call a *model pipeline*. Raven supports model pipelines expressed in a generic and portable model format [25] that is compatible with MLflow [24], and stores them in the RDBMS.

*The work was done while the author was at Microsoft.

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.

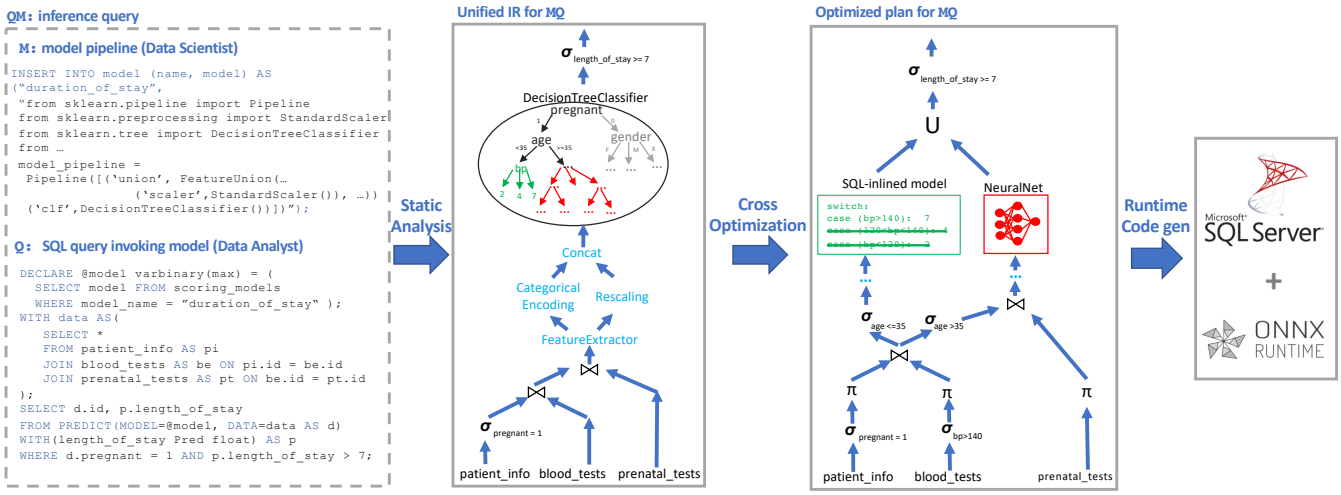

Figure 1: Running example: find pregnant patients with predicted length of stay in the hospital longer than a week.

Users can then invoke them (on data stored in the DB or on fresh data coming from an application) by issuing SQL commands. We term **inference query** a query that invokes a model pipeline.

As we show in the rest of this paper, delivering competitive performance for in-DB inference requires a substantial engineering and research effort. Our running example touches upon several of the interesting opportunities we discovered in doing so (§2). Raven introduces an intermediate representation (IR) that includes both ML and relational operators. Input inference queries are captured in this IR by means of static analysis (§3). The IR is then analyzed and optimized using novel cross-operator optimizations and transformations that Raven proposes (§4). Finally, the optimized IR is fed for execution to the DBMS that supports different ML runtimes. To achieve best-in-class performance for our optimized plans, we integrated ONNX Runtime[1] natively within SQL Server (§5). Given that support for native execution of all ML pipelines is elusive, Raven also employs out-of-process [14] and containerized execution [40] as required to achieve 100% coverage.

We show that (i) SQL Server with integrated ONNX Runtime is a solid building block for high-performance inference—yielding up to 5.5× speedups over standalone solutions; (ii) Raven's cross-optimizations yield benefits of up to 24× compared to unoptimized inference queries.

We have made an early preview of the ONNX Runtime integration available with Azure's SQL Database Edge [13], and our plan is to also make it available in other SQL Server offerings, both on-premises and in the cloud. While Raven is far from a finished system, the existing implementation already demonstrates the great potential for both research and industrial impact, by extending DBMSs with their robust capabilities to handle inference. We are busy incorporating the techniques we present in this paper in a full-fledged cost-based optimizer—hardware acceleration and multi-query optimization will make this even more fun.

[1]ONNX Runtime [31] is a state-of-the-art inference engine with support for diverse environments and backends, which we built and open-sourced at Microsoft. It supports all models that can be expressed in ONNX [30], i.e., the vast majority of models.

2. Raven OVERVIEW

Our running example is *predicting the duration of stay in a hospital*,[2] depicted in Fig. 1.

A data scientist has developed a decision tree model `M` that predicts a patient's length of stay in a hospital, by combining `patient_info` with results from `blood_tests` and `prenatal_tests`. The model is trained over large amounts of data (e.g., across all hospitals in an insurance network) and is deployed/stored within the RDBMS. At a later time, an analyst, employed by a specific hospital, issues a SQL query `Q` to apply the model on local data in order to "*find pregnant patients with a high likelihood of staying in the hospital for more than a week*" and inform the medical staff.

By storing and scoring the model within the RDBMS, we inherit ease of access via SQL, along with several desirable properties regarding updates to the deployed model: *transactionality* (a change to the model can be handled as part of a transaction), *high availability*, and *auditability*. To achieve good performance, Raven employs several optimizations and performs inference natively in the RDBMS, invoking an ML runtime as an integral part of the database runtime.

The input inference query `QM`, which includes both the SQL query `Q` and the model pipeline `M` (in Python here), is handled as follows. First, Raven's *Static Analyzer* parses `QM` and performs static analysis on the SQL and Python scripts. The result is a DAG expressed in Raven's unified IR (detailed in §3), as shown in Fig. 1.

The IR is fed to the *Cross Optimizer*, which performs various optimizations (passing information between the data and ML operators) and operator transformations. It also determines which part of the IR will be executed by SQL Server and which by the integrated ML runtime (ONNX Runtime here). The optimization space is very rich—below we provide some representative optimizations, which we further discuss in §4:

- *predicate-based model pruning*: the condition `pregnant=1` is pushed upward and into the decision tree, resulting in the right subtree being pruned.

[2]The example is based on [34], with changes designed to showcase several Raven optimizations.

- *model-projection pushdown*: unused or zero-weight features can be projected-out early in the query plan—this is common due to model regularization or due to the above pruning (e.g., `gender` is no longer used).
- *model/query splitting*: the pruned model can be partitioned in a cheap model (for `age<=35`) and a more complex one (for `age>35`). Model and query are thus split in two branches and separately optimized.[3]
- *model inlining*: small decision trees can be inlined thanks to SQL Server's recent UDF inlining feature [37].
- *NN translation*: Raven can transform many classical ML models (e.g., decision tree) and featurizers into equivalent neural networks (NN) to then leverage the highly optimized ONNX Runtime for batch scoring on CPU/GPU.
- *standard DB optimizations*: such as predicate/projection pushdown and join elimination can be triggered—in the inlined left-branch we don't need to join with `prenatal_tests`, and `bp>140` can be derived and pushed-down.
- *compiler optimizations*: we implemented compiler-style optimizations such as constant-folding within ONNX Runtime—the `pregnant` variable is a constant in our example query and can be propagated inside the NN.

The optimized Raven IR is passed to the *Runtime Code Generator*, which generates a new SQL query, reflecting the above optimizations. The integrated SQL Server+ONNX Runtime engine is then invoked for execution.[4]

It is clear from the above that extensive optimizations are possible once we bring ML inference into the DBMS. At the time of writing, we have added native support for ONNX Runtime within SQL Server. We have designed and implemented several of these optimizations, and automated the static analysis process. In the next sections, we describe the path we are taking towards building an optimizer and runtime for integrated evaluation of inference queries.

3. Raven IR AND STATIC ANALYSIS

Intermediate representations have been commonly used for enabling optimizations in various settings. Most database query optimizers rely on relational algebra, whereas different IRs have been proposed for ML runtimes [43, 30].

In Raven, we chose to combine both data and ML operators in a *unified* IR, as shown in Fig. 1. This allows us to optimize an inference query that includes both data and ML operations in a holistic manner: we can perform optimizations that span data and ML operations, and pick the most suitable runtime to execute each operator (§4).

Next, we define Raven's IR and describe the static analysis process to extract the IR from an inference query.

3.1 Raven IR

Raven's data and ML operators are chosen to cover most practical scenarios, based on our analysis of ~4.6 million publicly available Python notebooks from GitHub [35]. Our current operator set, which is easily extensible, can be split into the following categories.

Relational algebra (RA). This includes all the relational algebra operators, which are found in a typical RDBMS.

Linear Algebra (LA). A large fraction of the operators used in ML frameworks, and in particular neural network runtimes [1, 31, 36], fall into this category. Examples include `matrix multiplication` and `convolution` operators.

Other ML operators and data featurizers (MLD). These are operators widely used in classical (non-NN) ML frameworks (e.g., scikit-learn [39], ML.NET [5]), but do not fall in the LA category, such as decision trees and featurization operations (e.g., categorical encoding, text featurization).

UDFs. When the static analyzer is not able to map part of the input into operators of the above categories (e.g., a function containing arbitrary Python code), a UDF operator is used to wrap the non-optimizable code as a black box.

Note that our IR includes both higher- and lower-level operators. For example, a linear regression operator (higher-level) can also be expressed as a set of linear algebra operators (lower-level). We purposely allow diverse operator levels to unlock different optimizations, similar to MLIR [26].

3.2 Static Analysis

An inference query consumed by Raven (see Fig. 1) is a SQL query that performs (part of) the data processing and invokes ML model pipelines.[5] The whole inference query can be instead expressed as a script in some imperative language (e.g., Python or R). The input scripts are accompanied by metadata to specify the required runtimes and dependencies (e.g., Python version, libraries used), and to access the referenced data and models. An open model format, such as the one defined in MLFlow [25], can be used for this purpose.

Translating the SQL part into the IR is straightforward (similar to a DB parser that builds a logical plan). The interesting part is analyzing the model scripts expressed in an imperative language. Our current prototype supports Python scripts and notebooks, given their popularity in ML [20].

Given a Python script, the Static Analyzer[6] performs lexing, parsing, extraction of variables and their scopes, semantic analysis, type inference, and finally extraction of control and data flows. To compile the dataflow to an equivalent IR plan, the Static Analyzer takes as input an in-house knowledge base of APIs of popular data science libraries (e.g., Pandas [33], NumPy [29], scikit-learn [39], PyTorch [36]), along with functions that map dataflow nodes/subgraphs to equivalent IR operators. Dataflow parts that cannot be translated to IR operators are translated to UDFs.

This static analysis process comes with several challenges and limitations (again, we use UDFs when we cannot overcome them). First, translating loops to relational or linear algebra operators is known to be a hard, if not undecidable, problem [4]. In our analysis of the ~4.6 millions Python notebooks, however, we found that only ~17% of all notebook code cells use such constructs. Thus, the vast majority of cases can be handled through analysis of straight line code blocks. Second, conditionals result in potentially multiple execution paths. In such cases, the Static Analyzer will ex-

[3]This shares commonalities with model cascades [21].

[4]For inference queries that are not yet supported by our static analysis or by ONNX Runtime, we support calling external ML runtimes and containerized execution.

[5]There is no standardized way yet to invoke models in SQL. Here we use the SQL Server way (as of version 2017) through the `PREDICT` or the `sp_execute_external_script` statements [28, 14].

[6]A full paper with detailed description is in the works along with plans to open source the Python static code analyzer.

tract one plan per execution path. Hence, downstream components in Raven need to operate based on multiple plans. Third, in dynamically typed languages, such as Python, type inference may result in assigning a collection of potential types to variables. We plan to use knowledge from the SQL part to improve type inference in many practical scenarios.

In most practical cases we tested, static analysis takes less than 10 msec. Its end result is a Raven IR plan that is given as input to the Cross Optimizer, discussed next.

4. CROSS-OPTIMIZATIONS

In this section, we focus on the novelty aspects of our optimizer: cross-IR optimizations that pass information between data and ML operators (§4.1), and transformations between operators to allow more efficient engines to be used for the same operation (§4.2). All our optimizations can be expressed as transformation rules, applied by Raven's Cross Optimizer (§4.3). By using state-of-the-art relational and ML engines, Raven can also leverage the large body of work in relational and ML inference optimization [31, 41, 11, 23]—we do not further discuss such techniques here.

While discussing each optimization, we also evaluate its benefits, using two datasets: (i) patient information to predict length of stay in hospital (per our running example in §2); and (ii) flight information to predict whether a flight will be delayed.[7] We use dataset sizes of up to 10M tuples for inference (1.25 GB on disk). Unless mentioned otherwise, all experiments are run on an Azure D16s_v3 VM instance, with 16 vCPUs, 64 GB of RAM, and a 1.1 TB SSD. Numbers are averages over multiple warm runs, and for each run we count the time it takes to load the model, perform the optimization, read the data, and perform inference over it.

4.1 Cross-IR optimizations

In this set of optimizations, we exploit ML operator characteristics (e.g., model weights) to optimize data operations of the inference query (*model-to-data*), or leverage relational operator- and data-properties to optimize the ML part of the query (*data-to-model*). Below we present some first such techniques we have devised—many more can be introduced.

These cross-IR optimizations can be seen as a form of Sideways Information Passing (SIP) [8]. However, unlike SIP that requires adapting physical operators, our techniques are applied purely at query optimization time.

Predicate-based model pruning. This data-to-model optimization exploits predicates in the IR (e.g., coming from the `WHERE` clause of the SQL query or a Pandas' filter) to simplify a model.

In our running example (Fig. 1), we can propagate the filter `pregnant=1` to the downstream decision tree model. The right branch of the tree can then be eliminated, thereby improving its prediction time—by 29% in our example.

Predicate-based pruning can also be beneficial for categorical features. Such features are typically encoded as a set of binary features, one for each unique value of the original feature. If there is a selection on the original feature, only one of the corresponding binary features will be non-zero. Hence, the rest of the features can be dropped from the model. We trained a logistic regression model for the flight delay and added a filter on the destination airport. Predicate-based pruning yields a ∼2.1× on this query using scikit-learn, regardless of the filter's selectivity (what matters in this speed-up is the number of features dropped).

Likewise, we can improve a neural network's performance via constant folding,[8] i.e., statically computing part of the model based on the constant input from the predicate.

This technique can also be applied based on data properties instead of explicit selections. Using data statistics, we might observe that only specific unique values appear in the data or that data follows specific distributions (e.g., all patients are above 35). In these cases, we can derive predicates to perform predicate-based pruning.

Model-projection pushdown. In this model-to-data optimization, we observe properties of an ML operator to simplify the data processing part of the inference query.

Consider a logistic or linear regression model with some of its weights being zero. This is often the case when L_1-regularization techniques (e.g. Lasso [42]) are applied during training to improve the model's generalization ability, size, and prediction cost. Here we exploit this property further. The features that will be multiplied with these zero-weights do not contribute to the prediction, and can therefore be projected out from the data part and be removed from the model without affecting the inference result.

We trained logistic regression models for flight delay, using scikit-learn and various L_1-regularization strengths.[9] We picked the two highest-performing models (i.e., with the highest AUC): the one had 41.75% sparsity (that is, percentage of zero weights), the other 80.96%. Fig. 2(a) shows that model-projection pushdown improves inference time by ∼1.7× for the first model and ∼5.3× for the second.

Even for inference queries that model-projection pushdown is not immediately applicable, it can be enabled by first applying other optimizations: in Fig. 1, predicate-based pruning of the right tree branch enables model-projection pushdown on `gender`, as this feature is no longer needed. Similarly, it can enable other optimizations: after eliminating features, the relational optimizer can drop joins if one of the joining relations no longer provides features needed by the model.

There are several more questions we plan to investigate: What is the impact of physical database design, such as column stores, when applying model-projection pushdown? What is the benefit for more complex models, such as NNs? What would be the impact in runtime and model accuracy when applying *lossy* model-projection pushdown where small, but non-zero, weights are removed?

Model clustering. Taking predicate-based pruning using data properties a step further, we may not have a single value for one or more features, but we could cluster the data in a way that each cluster has specific values for some features. We can then precompile optimized models for each cluster.

We performed k-means clustering with an increasing number of clusters for 700K tuples of flight delay. Fig. 2(b) shows that model clustering reduces inference time by up to 54%. The more the clusters the bigger the gain, although the relative gain diminishes after a point. Model compile time, i.e., the time to create new models by dropping features, is negligible. On the other hand, hospital stay does not bene-

[7] https://www.kaggle.com/usdot/flight-delays

[8] https://github.com/microsoft/onnxruntime/blob/master/onnxruntime/core/optimizer/constant_folding.h

[9] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

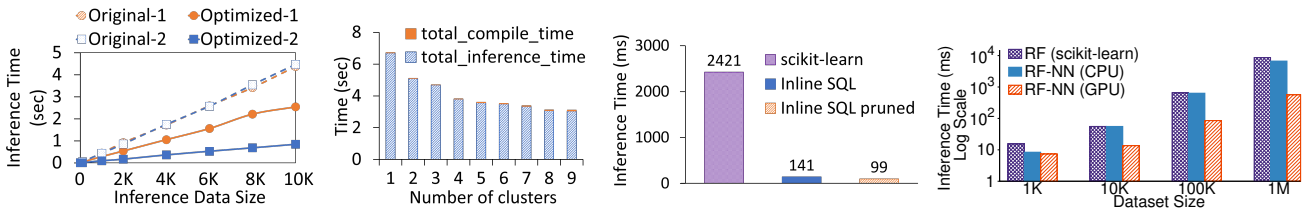

Figure 2: Left to right: cross-optimizations for flight delay ((a) model-projection pushdown and (b) model clustering) and operator transformations for hospital stay ((c) model inlining and (d) NN translation).

fit from clustering since its categorical features are already binary, therefore fewer features are dropped.

Clustering can be relatively expensive, depending on input data size (0.4 to 42 secs in our examples). In practical settings, clustering can be performed offline on a sample of historical data. When new data arrives, we use the precompiled models whenever possible; if such a model does not exist, we fall back to the original model.

4.2 Operator Transformations

Along with the logical optimizations presented above, Raven applies rules that transform (a set of) operator(s) to another. For example, we can map a linear regression to a matrix multiplication.

Such transformations enable both additional optimizations and the use of different runtimes for executing an operator (e.g., a high-performance NN engine might not have a dedicated linear regression, but would support the lower-level operator it got translated to). Note that transformations should preserve semantics (e.g., SQL's bag semantics vs. Pandas' ordered bags).

Model inlining. These transformations translate ML operators (LA and MLD operators, see §3) to relational ones. Several of these transformations have been studied in the literature [18, 2, 10, 38]. They are particularly important in Raven, because they allow us to use the relational optimizations and high performance of SQL Server for data operations (e.g., to execute a join that was initially expressed in Python). Moreover, we employ the UDF inlining technique introduced in SQL Server 2019 [37] to further boost performance.

We trained a decision tree (the same technique would work for tree ensembles) for the hospital stay in scikit-learn, translated it to a UDF after expressing its conditions as a SQL query, and inlined the UDF in the query. Fig. 2(c) demonstrates that this ML-to-relational operator translation yields a performance gain of ~17× for a dataset of 300K tuples when compared to running the decision tree in scikit-learn reading data from the DB (reading from a CSV was similar). Big part of this gain was due to completely avoiding data transformations by keeping execution inside the DB. Assuming a query with a selection on a tree's dimension, as discussed above, we can further improve runtime by 29% with predicate-based pruning, leading to a total improvement of 24.5×.

We also experimented with pushing categorical encodings to SQL Server. Our initial experiments show significant performance improvements when the number of resulting features is not too big, but further investigation is required to draw safe conclusions.

NN translation. Raven introduces novel transformations from MLD operators (see §3) to linear algebra ones [27]. This allows us to express classical ML operators and data featurizers, typically written in frameworks like scikit-learn and ML.NET, to neural-networks that can be executed in highly efficient engines like ONNX Runtime, PyTorch, and TensorFlow. This is very important performance-wise: unlike most traditional ML frameworks, NN engines support out-of-the-box hardware acceleration through GPUs/FPGAs/NPUs, as well as code generation [11][10].

In Fig. 2(d), we compare a random forest model (RF) for hospital stay in scikit-learn against the NN translation of the same model (RF-NN), both on CPU and GPU. Here we used a machine with similar specs to our VM but equipped with an Nvidia K80 GPU. For 1K tuples, RF-NN on CPU is about 2× faster compared to the RF on scikit-learn, whereas RF-NN on GPU further decreases computation time by 10%. As we increase the dataset size, the gap between scikit-learn and RF-NN on CPU closes, and both have performance increasing almost linearly to the dataset size. Conversely, with larger datasets we can better utilize the parallel architecture of the GPU, and therefore get a speed-up of up to 15× compared to scikit-learn for 1M tuples.

4.3 Raven's Cross Optimizer

So far, we discussed various transformation rules (cross-optimizations and operator transformations) that we have introduced and implemented in Raven, and showed their benefits on real models/data. The next important step in our journey is to integrate all these rules in our optimizer—we are actively working on this at the time of writing. An initial version will be heuristic-based, applying all rules in a specific order. Our goal is to then get to a cost-based Cascades-style optimizer [16], possibly integrated with SQL Server's optimizer, in which each operator will be associated with a cost. Several plan alternatives will be considered by applying the rules in different orders and the best will be picked. Note that as part of the optimization process, we need to pick the runtime that each operator will be executed on (relational engine or ML runtime), based on each runtime's capabilities and performance (including specialized hardware) and the cost of switching across engines.

5. INFERENCE QUERY EXECUTION

Raven's Runtime Code Generator builds a new SQL query that corresponds to the optimized IR (i.e., the output of the Cross Optimizer). The model invocations that are included in the optimized SQL query will be executed in one of the

[10]We are also working on similar translations from traditional ML pipelines to neural networks for the model training side [44].

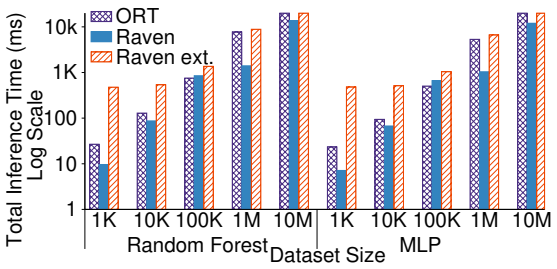

Figure 3: Model inference performance for SQL Server with in- and out-of-process ONNX Runtime (Raven and Raven Ext, resp.) and standalone ONNX Runtime (ORT).

following ways, in decreasing level of integration with SQL Server's main relational engine:

In-process execution (Raven). Starting with version 2017, SQL Server introduced the `PREDICT` statement [28] to allow native inference for a small set of five models (such as linear and logistic regression, and decesion trees). As part of realizing our vision, we deeply integrated ONNX Runtime inside SQL Server. ONNX Runtime is used as a dynamically linked library to create inference sessions, transform data to tensors, and invoke in-process predictions over any ONNX model or any model that can be expressed in ONNX through Raven's static analysis or ONNX converters [32]. A user simply needs to store their model in SQL Server and issue queries that include model inference using the existing `PREDICT` statement. This is the tightest-integration option: apart from in-process execution for performance (e.g., by avoiding unnecessary data copies/conversions) and security purposes (by not letting data leave the process boundaries), it allows us, out-of-the-box, to take advantage of model and inference-session caching and of SQL Server's optimizer.

Out-of-process execution (Raven Ext). For model pipelines not yet supported by our Static Analyzer, we use the `sp_execute_external_script` [14] statement, which instantiates an external language runtime to perform out-of-process inference. Currently supported languages are Python and R (and Java with SQL Server 2019). Although not as tightly integrated, this mode of execution has the advantage of not requiring any changes to the SQL Server code—any existing SQL Server installation with support for external scripts can take advantage of it.

Containerized execution. For model pipelines that cannot be executed with one of the above techniques (i.e., written in a language that is supported neither by our Static Analyzer nor by `sp_execute_external_script`), we fall back to spinning up a Docker container and performing inference through a REST endpoint [40].

Having shown the substantial benefits of our optimizations in §4, we turn to the following question: *Can our in-process integration of ONNX Runtime with SQL Server match the performance of a standalone ONNX Runtime instance when performing pure model inference (no SQL part)?* Or are there significant overheads in the integration?

We compare: (i) standalone ONNX Runtime (*ORT* hereafter), (ii) Raven, and (iii) Raven Ext. We use both a random forest (RF) and a multi-layer perceptron (MLP) as part of a pipeline that also includes featurization, and translate both end-to-end pipelines to NNs to be efficiently executed in ORT (see NN translation, §4). Fig. 3 shows results for increasing dataset size.

We make the following observations:
(i) Between 50K and 100K tuples, ORT and Raven have similar performance, with Raven having an overhead of up to 15%. We are positive that we can further close this gap with implementation improvements. That said, this overhead is insignificant, compared to the improvements of orders of magnitude that Raven's optimizations provide.[11]
(ii) For smaller datasets (e.g., up to 50K tuples) and warm runs, Raven is faster than ORT (e.g., 3msec vs. 20msec for 100 tuples). This is due to SQL Server's model and inference-session caching across queries, instead of loading the model from disk and relying on the file system cache.
(iii) For 1M and 10M tuples Raven is faster than ORT by around 5×! This came to our surprise. After investigation, we observed that for larger datasets, SQL Server *automatically* parallelizes both the scan and `PREDICT` operators. When forcing sequential execution, Raven was about 7% slower than ORT. This model inference could potentially be parallelized in ORT as well, but that would not be trivial, whereas it came for free with SQL Server.
(iv) Raven Ext has a constant overhead of about half a second to start the external language runtime and some additional overheads, most probably due to data transfers. Still it is a viable option in cases our Static Analyzer does not support the model pipeline or for SQL Server installations that do not support in-process execution. We are currently investigating how the overheads of out-of-process execution could be reduced.
(v) In our implementation of Raven, we gained about an order of magnitude by performing batch inference instead of one prediction per tuple (ideal batch size to be investigated).

6. RELATED WORK

Several previous works have proposed to run machine learning in the RDBMS [17, 19, 10, 2, 6]. Interestingly enough, these works have largely focused on model training, whereas the prime focus of Raven is inference of already trained machine learning models.

Existing approaches for model inference follow either a containerized [12] or an in-application [5] model. More recently, Amazon Aurora enabled external calls from SQL queries to machine learning models in SageMaker, which would also qualify as a containerized execution approach [7]. Although containerized execution allows for easy integration of ML runtimes with SQL engines, it introduces overheads that do not allow for low latency predictions. Google's BigQuery ML [9] is closer to Raven (although it relies mostly on hardcoded models), but targets mainly batch predictions, since it inherits the relatively high startup cost of queries in BigQuery. Instead, using SQL Server along with model caching allows Raven to perform even single-tuple predictions with low latency. Moreover, none of these engines take advantage of a unified representation of data and ML operators, which enables Raven to perform novel cross-optimizations that yield significant performance benefits.

Cross-optimization of relational and linear algebra operators is recently becoming a hot topic [18, 22], whereas spe-

[11] Some of our optimizations could be applied to an ML runtime too. However, they would lack the ability of a relational engine to perform data operations like joins and aggregates in a highly optimized fashion, as well as the other benefits of an RDBMS §1.

cific optimizers [23, 21] and runtimes [30, 31, 11] for model inference are also emerging. Our goal with Raven is to bridge the gap between the two worlds: we propose an optimizer able to execute both runtime-specific and cross-IR optimizations in an end-to-end fashion.

7. CONCLUSION

We presented Raven, a system we are building to perform in-DB ML inference. Raven performs static analysis of Python ML pipelines and SQL queries, which are captured in a unified IR. This enables us to apply novel cross-optimizations, yielding performance gains of up to 24×. The target execution environment for this optimized IR is a deeply integrated version of SQL Server and ONNX Runtime, which alone provides up to 5.5× performance gains over standalone ONNX Runtime execution. The version of SQL Server integrated with ONNX Runtime is currently available in public preview in Azure's SQL Database Edge [13], and our plan is to bring such capabilities to other flavors of SQL Server databases, both on-premises and in the cloud. This is only the beginning of a long journey to incorporate ML scoring as a foundational extension of relational algebra and an integral part of SQL query optimizers and runtimes.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. LevelHeaded: A unified engine for business intelligence and linear algebra querying. In *ICDE*, 2018.

[3] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Gowdal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, J. Leeka, K. Park, H. Patel, O. Poppe, F. Psallidas, R. Ramakrishnan, A. Roy, K. Saur, R. Sen, M. Weimer, T. Wright, and Y. Zhu. Cloudy with high chance of DBMS: A 10-year prediction for enterprise-grade ML. In *CIDR*, 2020.

[4] M. B. S. Ahmad and A. Cheung. Automatically leveraging MapReduce frameworks for data-intensive applications. In *SIGMOD*, 2018.

[5] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W. Chin, Y. Dekel, X. Dupré, V. Eksarevskiy, S. Filipi, T. Finley, A. Goswami, M. Hoover, S. Inglis, M. Interlandi, N. Kazmi, G. Krivosheev, P. Luferenko, I. Matantsev, S. Matusevych, S. Moradi, G. Nazirov, J. Ormont, G. Oshri, A. Pagnoni, J. Parmar, P. Roy, M. Z. Siddiqui, M. Weimer, S. Zahirazami, and Y. Zhu. Machine learning at Microsoft with ML.NET. In *SIGKDD*, 2019.

[6] G. Alonso, Z. Istvan, K. Kara, M. Owaida, and D. Sidler. doppioDB 1.0: machine learning inside a relational engine. *IEEE Data Eng. Bull.*, 2019.

[7] Amazon Aurora Machine Learning. http://aws.amazon.com/rds/aurora/machine-learning, 2019.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, 1987.

[9] BigQuery ML. http://cloud.google.com/bigquery-ml, 2019.

[10] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative machine learning on Spark. *PVLDB*, 2016.

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: a low-latency online prediction serving system. In *NSDI*, 2017.

[13] Deploy and make predictions with an ONNX model in SQL Database Edge Preview. http://docs.microsoft.com/en-us/azure/sql-database-edge/deploy-onnx, 2019.

[14] Execute external scripts in SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-execute-external-script-transact-sql?view=sql-server-2017, 2019.

[15] GDPR. https://eugdpr.org/, 2019.

[16] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 1995.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 2012.

[18] D. Hutchison, B. Howe, and D. Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In *BeyondMR@SIGMOD 2017*, 2017.

[19] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *PVLDB*, 2019.

[20] Kaggle: The State of Data Science. http://www.kaggle.com/surveys/2017, 2019.

[21] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia. Willump: A statistically-aware end-to-end optimizer for machine learning inference. *CoRR*, abs/1906.01974, 2019.

[22] A. Kunft, A. Katsifodimos, S. Schelter, and V. Markl. An intermediate representation for optimizing machine learning pipelines. *PVLDB*, 2019.

[23] Y. Lee, A. Scolari, B. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, 2018.

[24] MLflow. http://mlflow.org, 2019.

[25] MLflow models. https://mlflow.org/docs/latest/models.html, 2019.

[26] MLIR. https://github.com/tensorflow/mlir, 2019.

[27] S. Nakandala, G. Yu, M. Weimer, and M. Interlandi. Compiling classical ml pipelines into tensor computations for one-size-fits-all prediction serving. *Systems for ML workshop at NeurIPS*, 2019.

[28] Native scoring with PREDICT statement in SQL Server. http://docs.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-2017, 2019.

[29] NumPy. http://www.numpy.org, 2019.

[30] ONNX. http://onnx.ai, 2019.

[31] ONNX Runtime. http://github.com/microsoft/onnxruntime, 2019.

[32] ONNXMLTools. http://github.com/onnx/onnxmltools, 2019.

[33] Pandas. http://pandas.pydata.org, 2019.

[34] Predicting Hospital Length of Stay. http://microsoft.github.io/r-server-hospital-length-of-stay, 2019.

[35] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. Data science through the looking glass and what we found there. *ArXiv e-prints*, 1912.09536, 2019. URL: https://arxiv.org/abs/1912.09536.

[36] PyTorch. http://pytorch.org, 2019.

[37] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham. Froid: optimization of imperative programs in a relational database. *PVLDB*, 2017.

[38] M. Schüle, M. Bungeroth, D. Vorona, A. Kemper, S. Günnemann, and T. Neumann. ML2SQL - compiling a declarative machine learning language to SQL and Python. In *EDBT*, 2019.

[39] scikit-learn. http://scikit-learn.org, 2019.

[40] SQL Server Big Data Clusters. http://docs.microsoft.com/en-us/sql/big-data-cluster/big-data-cluster-overview, 2019.

[41] TensorFlow Model Optimization. http://www.tensorflow.org/lite/performance/model_optimization, 2019.

[42] R. Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society (Series B)*, 58, 1996.

[43] XLA. http://www.tensorflow.org/xla, 2019.

[44] G. Yu, S. Amizadeh, A. Pagnoni, B. Chun, M. Weimer, and M. Interlandi. Making classical machine learning pipelines differentiable: A neural translation approach. *CoRR*, abs/1906.03822, 2019.