DO WE STILL NEED PEOPLE TO WRITE DATABASE SYSTEMS?

OSACON 2021

Carnegie Mellon University
OTTER TUNE
@andy_pavlo

#1 – Last 20 Years
#2 – Current ML Seduction
#3 – Next 20 Years

Specialized DBMSs for analytics have been around since the 1970s.

The OLAP DBMS landscape flourished in the 2000s because more organizations have large data sets than ever before.

Specialized DBMSs for analytics around since the 1970s.

The OLAP DBMS landscape flou 2000s because more organizati data sets than ever before.

"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

1. Introduction

Relational DBMSs arrived on the scene as research prototypes in the 1970's, in the form of System R [10] and INGRES [27]. The main thrust of both prototypes was to surpass IMS in value to customers on the applications that IMS was used for, namely "business data processing". Hence, both systems were architected for on-line transaction processing (OLTP) applications, and their commercial counterparts (i.e., DB2 and INGRES, respectively) found acceptance in this arena in the 1980's. Other vendors (e.g., Sybase, Oracle, and Informix) followed the same basic DBMS model, which stores relational tables row-by-row, uses B-trees for indexing, uses a cost-based optimizer, and provides ACID transaction properties.

Since the early 1980's, the major DBMS vendors have steadfastly stuck to a "one size fits all" strategy, whereby they maintain a single code line with all DBMS services. The reasons for this choice are straightforward — the use of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section 3, we discuss stream processing applications and indicate a particular example where a specialized stream processing engine outperforms an RDBMS by two orders of magnitude. Section 4 then turns to the reasons for the performance difference, and indicates that DBMS technology is not likely to be able to adapt to be competitive in this market. Hence, we expect stream processing engines to thrive in the marketplace. In Section 5, we discuss a collection of other markets where one size is not likely to fit all, and other specialized database systems may be feasible. Hence, the fragmentation of the DBMS market may be fairly extensive. In Section 6, we offer some comments about the factoring of system software into products. Finally, we close the paper with some concluding remarks in Section 7.

2. Data warehousing

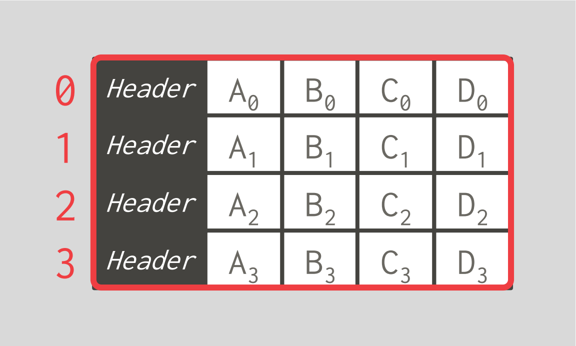
In the early 1990's, a new trend appeared: Enterprises wanted to gather together data from multiple operational databases into a data warehouse for business intelligence

2000s

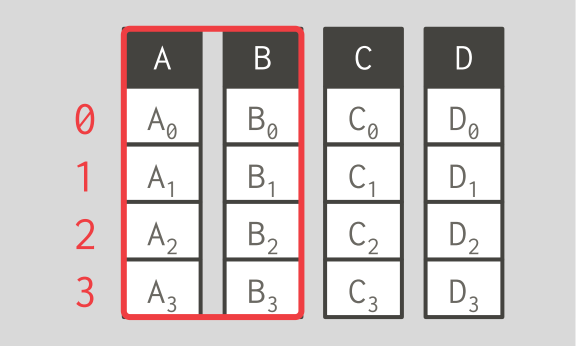
Columnar Storage

```
SELECT COUNT(B)
  FROM XXX
 WHERE A > ?;
```

Row Store



Column Store

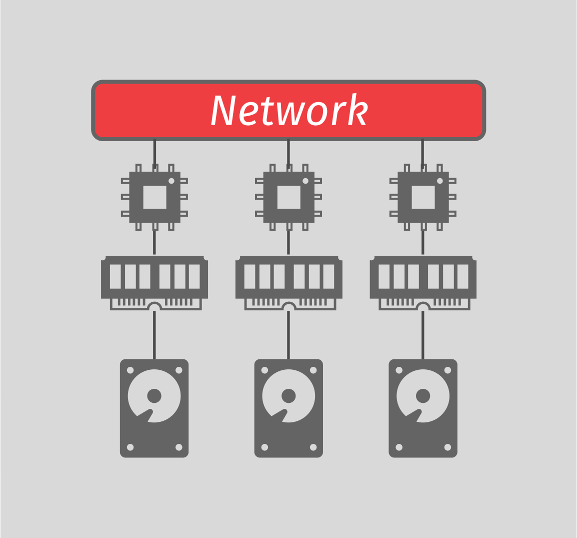


2000s

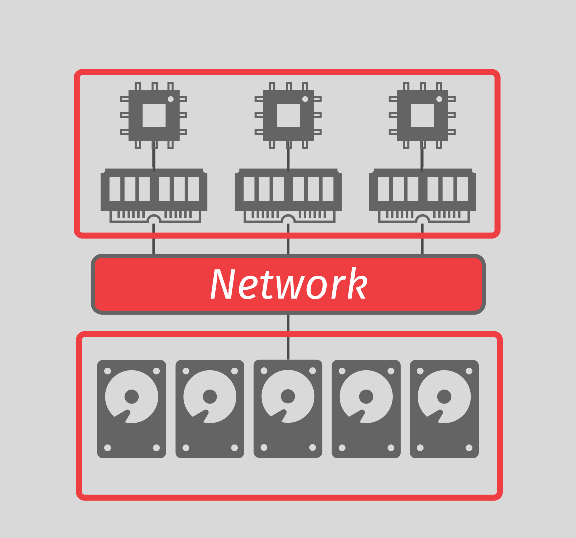
Columnar Storage

Disaggregated Storage

Shared Nothing



Shared Disk



2000s

Columnar Storage

Disaggregated Storage

Vectorized Execution

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
      ⮡(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

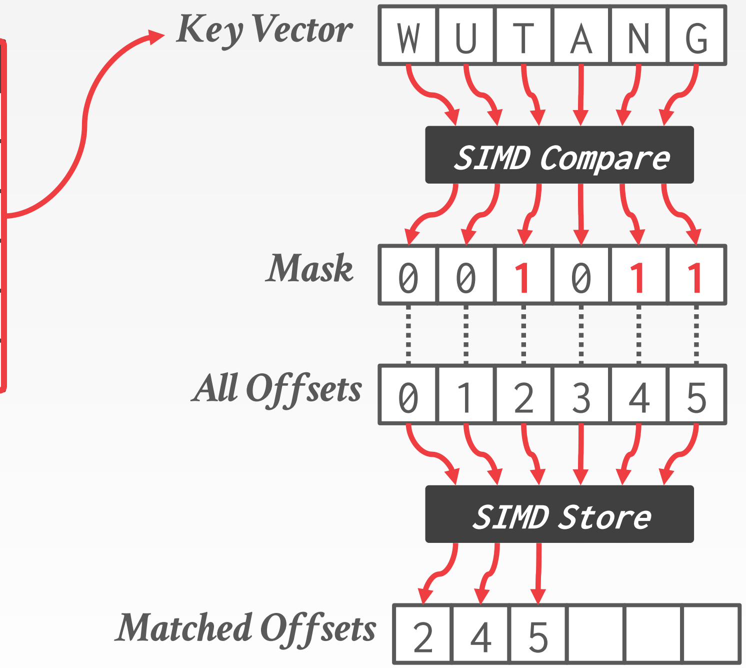
```
SELECT * FROM table
 WHERE key >= "G" AND key <= "T"
```

2000s

- Columnar Storage
- Disaggregated Storage
- Vectorized Execution

```
SELECT * FROM table
 WHERE key >= "G" AND key <= "T"
```

ID	KEY
1	W
2	U
3	T
4	A
5	N
6	G

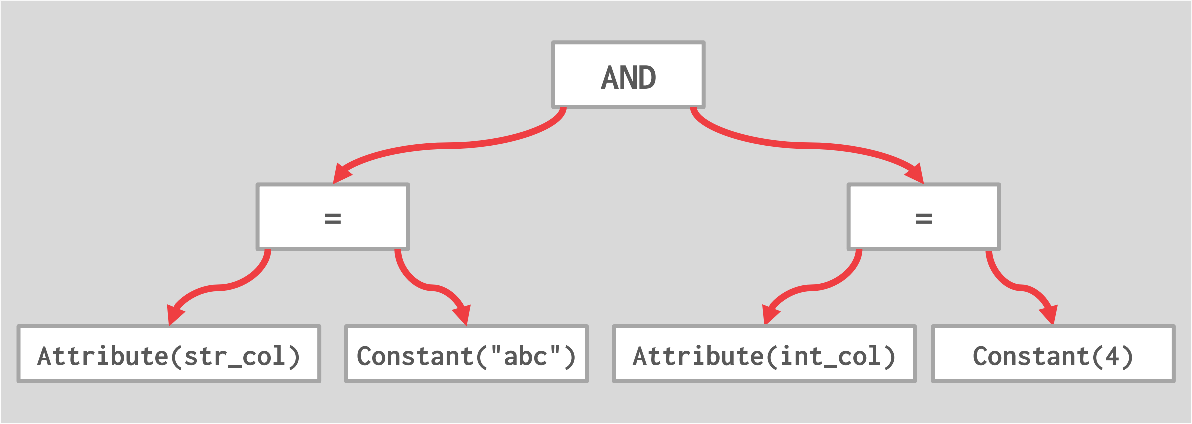


2010s

JIT Query Compilation

```sql
SELECT * FROM foo
 WHERE str_col = 'abc'
   AND int_col = 4;
```

Expression Tree



2010s

JIT Query Compilation

```
SELECT * FROM foo
 WHERE str_col = 'abc'
   AND int_col = 4;
```

Code Generated Plan

```
bool sel_eq_row(string str_col, string val0,
                int int_col, int val1) {
  return (str_col == val0 && int_col == val1);
}
```

2010s

JIT Query Compilation

UDF Inlining

TSQL Scalar functions are evil.

I've been working with a number of clients recently who all have suffered at the hands of TSQL Scalar functions. Scalar functions were introduced in SQL 2000 as a means to wrap logic so we benefit from code reuse and simplify our queries. Who would be daft enough not to think this was a good idea. I for one jumped on this initially thinking it was a great thing to do.

However as you might have gathered from the title scalar functions aren't the nice friend you may think they are.

If you are running queries across large tables then this may explain why you are getting poor performance.

In this post we will look at a simple padding function, we will be creating large volumes to emphasize the issue with scalar udfs.

```
create function PadLeft(@val varchar(100), @len int, @char char(1))
returns varchar(100)
as
begin
  return right(replicate(@char,@len) + @val, @len)
end
go
```

Interpreted

Scalar functions are interpreted code that means EVERY call to the function results in your code being interpreted. That means overhead for processing your function is proportional to the number of rows.

Running this code you will see that the native system calls take considerable less time than the UDF calls. On my machine it takes 2614 ms for the system calls and 38758ms for the UDF. Thats a 19x increase.

```
set statistics time on
go
select max(right(replicate('0',100) + o.name + c.name, 100))
from msdb.sys.columns o
cross join msdb.sys.columns c

select max(dbo.PadLeft(o.name + c.name, 100,'0'))
from msdb.sys.columns o
cross join msdb.sys.columns c
```

Source: Karthik Ramachandra

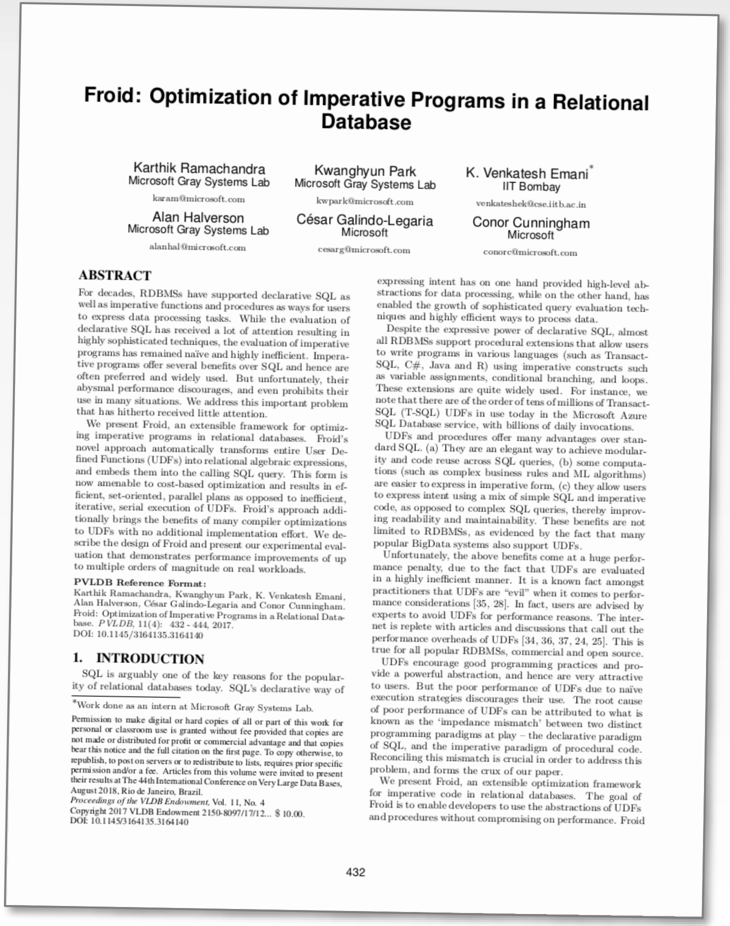
2010s

JIT Query Compilation

UDF Inlining

```
CREATE FUNCTION getVal(@x int)
RETURNS char(10) AS
BEGIN
 DECLARE @val char(10);
 IF (@x > 1000)
  SET @val = 'high';
 ELSE
  SET @val = 'low';
 RETURN @val + ' value';
END
```

```
SELECT getVal(5000);
```



Froid: Optimization of Imperative Programs in a Relational Database

Karthik Ramachandra, Microsoft Gray Systems Lab, karam@microsoft.com
Kwanghyun Park, Microsoft Gray Systems Lab, kwpark@microsoft.com
K. Venkatesh Emani*, IIT Bombay, venkateshek@cse.iitb.ac.in
Alan Halverson, Microsoft Gray Systems Lab, alanhal@microsoft.com
César Galindo-Legaria, Microsoft, cesarg@microsoft.com
Conor Cunningham, Microsoft, conorc@microsoft.com

ABSTRACT

For decades, RDBMSs have supported declarative SQL as well as imperative functions and procedures as ways for users to express data processing tasks. While the evaluation of declarative SQL has received a lot of attention resulting in highly sophisticated techniques, the evaluation of imperative programs has remained naïve and highly inefficient. Imperative programs offer several benefits over SQL and hence are often preferred and widely used. But unfortunately, their abysmal performance discourages, and even prohibits their use in many situations. We address this important problem that has hitherto received little attention.

We present Froid, an extensible framework for optimizing imperative programs in relational databases. Froid's novel approach automatically transforms entire User Defined Functions (UDFs) into relational algebraic expressions, and embeds them into the calling SQL query. This form is now amenable to cost-based optimization and results in efficient, set-oriented, parallel plans as opposed to inefficient, iterative, serial execution of UDFs. Froid's approach additionally brings the benefits of many compiler optimizations to UDFs with no additional implementation effort. We describe the design of Froid and present our experimental evaluation that demonstrates performance improvements of up to multiple orders of magnitude on real workloads.

PVLDB Reference Format:
Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria and Conor Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB*, 11(4): 432 - 444, 2017.
DOI: 10.1145/3164135.3164140

1. INTRODUCTION

SQL is arguably one of the key reasons for the popularity of relational databases today. SQL's declarative way of expressing intent has on one hand provided high-level abstractions for data processing, while on the other hand, has enabled the growth of sophisticated query evaluation techniques and highly efficient ways to process data.

Despite the expressive power of declarative SQL, almost all RDBMSs support procedural extensions that allow users to write programs in various languages (such as Transact-SQL, C#, Java and R) using imperative constructs such as variable assignments, conditional branching, and loops. These extensions are quite widely used. For instance, we note that there are of the order of tens of millions of Transact-SQL (T-SQL) UDFs in use today in the Microsoft Azure SQL Database service, with billions of daily invocations.

UDFs and procedures offer many advantages over standard SQL. (a) They are an elegant way to achieve modularity and code reuse across SQL queries, (b) some computations (such as complex business rules and ML algorithms) are easier to express in imperative form, (c) they allow users to express intent using a mix of simple SQL and imperative code, as opposed to complex SQL queries, thereby improving readability and maintainability. These benefits are not limited to RDBMSs, as evidenced by the fact that many popular BigData systems also support UDFs.

Unfortunately, the above benefits come at a huge performance penalty, due to the fact that UDFs are evaluated in a highly inefficient manner. It is a known fact amongst practitioners that UDFs are "evil" when it comes to performance considerations [35, 28]. In fact, users are advised by experts to avoid UDFs for performance reasons. The internet is replete with articles and discussions that call out the performance overheads of UDFs [34, 36, 37, 24, 25]. This is true for all popular RDBMSs, commercial and open source.

UDFs encourage good programming practices and provide a powerful abstraction, and hence are very attractive to users. But the poor performance of UDFs due to naïve execution strategies discourages their use. The root cause of poor performance of UDFs can be attributed to what is known as the 'impedance mismatch' between two distinct programming paradigms at play – the declarative paradigm of SQL, and the imperative paradigm of procedural code. Reconciling this mismatch is crucial in order to address this problem, and forms the crux of our paper.

We present Froid, an extensible optimization framework for imperative code in relational databases. The goal of Froid is to enable developers to use the abstractions of UDFs and procedures without compromising on performance. Froid

*Work done as an intern at Microsoft Gray Systems Lab.

Proceedings of the VLDB Endowment, Vol. 11, No. 4

DOI: 10.1145/3164135.3164140

Source: Karthik Ramachandra

2010s

JIT Query Compilation

UDF Inlining

```
CREATE FUNCTION getVal(@x int)
RETURNS char(10) AS
BEGIN
 DECLARE @val char(10);
 IF (@x > 1000)
  SET @val = 'high';
 ELSE
  SET @val = 'low';
 RETURN @val + ' value';
END
```

```
SELECT getVal(5000);
```

```
SELECT returnVal FROM
 (SELECT CASE WHEN @x > 1000
       THEN 'high'
       ELSE 'low' END AS val)
 AS DT1
OUTER APPLY
 (SELECT DT1.val + ' value'
        AS returnVal) DT2
```

Froid Inlining

```
SELECT returnVal FROM
 (SELECT 'high' AS val)
 AS DT1
 OUTER APPLY
 (SELECT DT1.val +
        ' value'
        AS returnVal)
 AS DT2
```

Dynamic Slicing

```
SELECT returnVal FROM
 (SELECT 'high value'
         AS returnVal)
 AS DT1
```

Const Propagation & Folding

```
SELECT 'high value';
```

Dead Code Elimination

Source: Karthik Ramachandra

2010s

JIT Query Compilation

UDF Inlining

```
CREATE FUNCTION getVal(@x int)
RETURNS char(10) AS
BEGIN
 DECLARE @val char(10);
 IF (@x > 1000)
  SET @val = 'high';
 ELSE
  SET @val = 'low';
 RETURN @val + ' value';
END
```

```
SELECT getVal(5000);
```

```
SELECT 'high value';
```

```
SELECT returnVal FROM
 (SELECT CASE WHEN @x > 1000
      THEN 'high'
      ELSE 'low' END AS val)
 AS DT1
OUTER APPLY
 (SELECT DT1.val + ' value'
        AS returnVal) DT2
```

Froid Inlining

```
SELECT returnVal FROM
 (SELECT 'high' AS val)
 AS DT1
 OUTER APPLY
 (SELECT DT1.val +
        ' value'
        AS returnVal)
 AS DT2
```

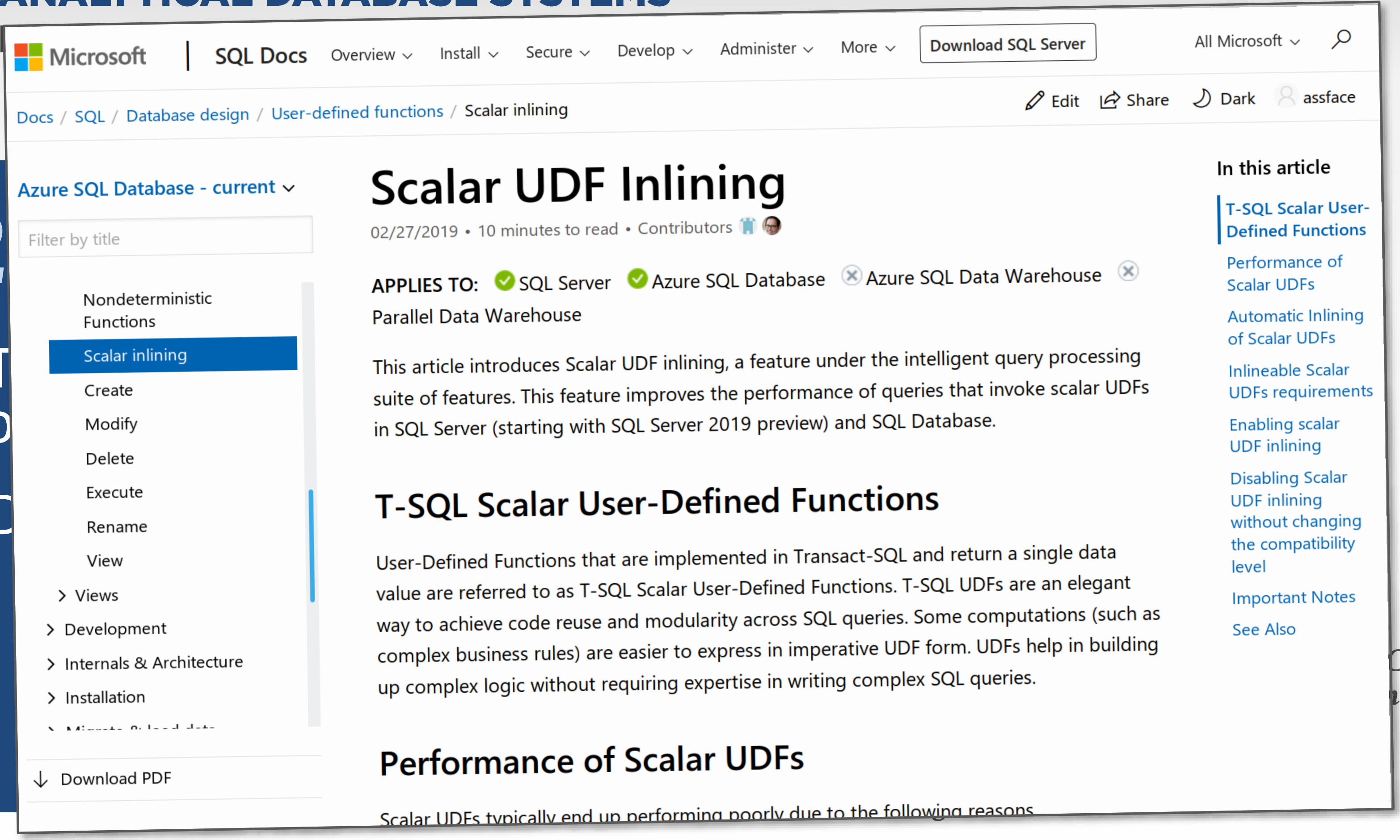
Dynamic Slicing

```
SELECT returnVal FROM
 (SELECT 'high value'
         AS returnVal)
 AS DT1
```

Const Propagation & Folding

Dead Code Elimination

Source: Karthik Ramachandra



Microsoft | SQL Docs Overview Install Secure Develop Administer More Download SQL Server All Microsoft

Docs / SQL / Database design / User-defined functions / Scalar inlining

Edit Share Dark assface

Azure SQL Database - current

Filter by title

- Nondeterministic Functions
- Scalar inlining
- Create
- Modify
- Delete
- Execute
- Rename
- View
- Views
- Development
- Internals & Architecture
- Installation

Download PDF

Scalar UDF Inlining

02/27/2019 • 10 minutes to read • Contributors

APPLIES TO: SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This article introduces Scalar UDF inlining, a feature under the intelligent query processing suite of features. This feature improves the performance of queries that invoke scalar UDFs in SQL Server (starting with SQL Server 2019 preview) and SQL Database.

T-SQL Scalar User-Defined Functions

User-Defined Functions that are implemented in Transact-SQL and return a single data value are referred to as T-SQL Scalar User-Defined Functions. T-SQL UDFs are an elegant way to achieve code reuse and modularity across SQL queries. Some computations (such as complex business rules) are easier to express in imperative UDF form. UDFs help in building up complex logic without requiring expertise in writing complex SQL queries.

Performance of Scalar UDFs

Scalar UDFs typically end up performing poorly due to the following reasons

In this article

- T-SQL Scalar User-Defined Functions
- Performance of Scalar UDFs
- Automatic Inlining of Scalar UDFs
- Inlineable Scalar UDFs requirements
- Enabling scalar UDF inlining
- Disabling Scalar UDF inlining without changing the compatibility level
- Important Notes
- See Also

Source: Karthik Ramachandra

2020s

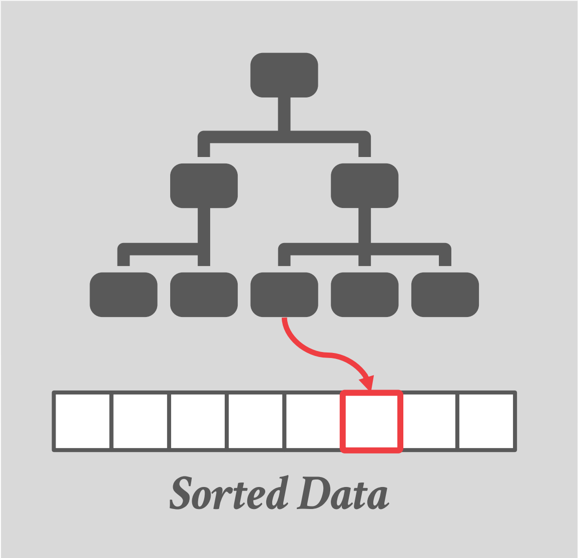
Learned Components

A learned component is an implemented portion of a DBMS that uses ML on previous observations to determine its future behavior as opposed a human-devised strategy.

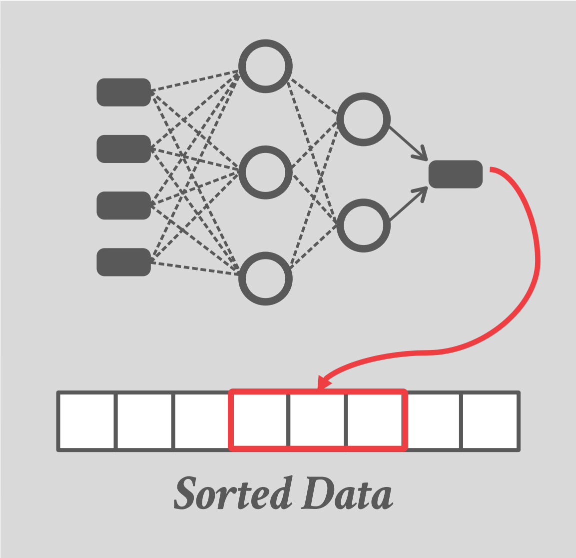
2020s

Learned Components

Traditional Index



Learned Index



Execution

- *Indexes*
- *Sorting Algorithms*
- *Hashing Algorithms*
- *Scheduling*

Query Planning

- *Cardinality Estimation*
- *Cost Models*
- *Join Ordering Search*
- *SQL Rewriting*
- *Predicate Inference*

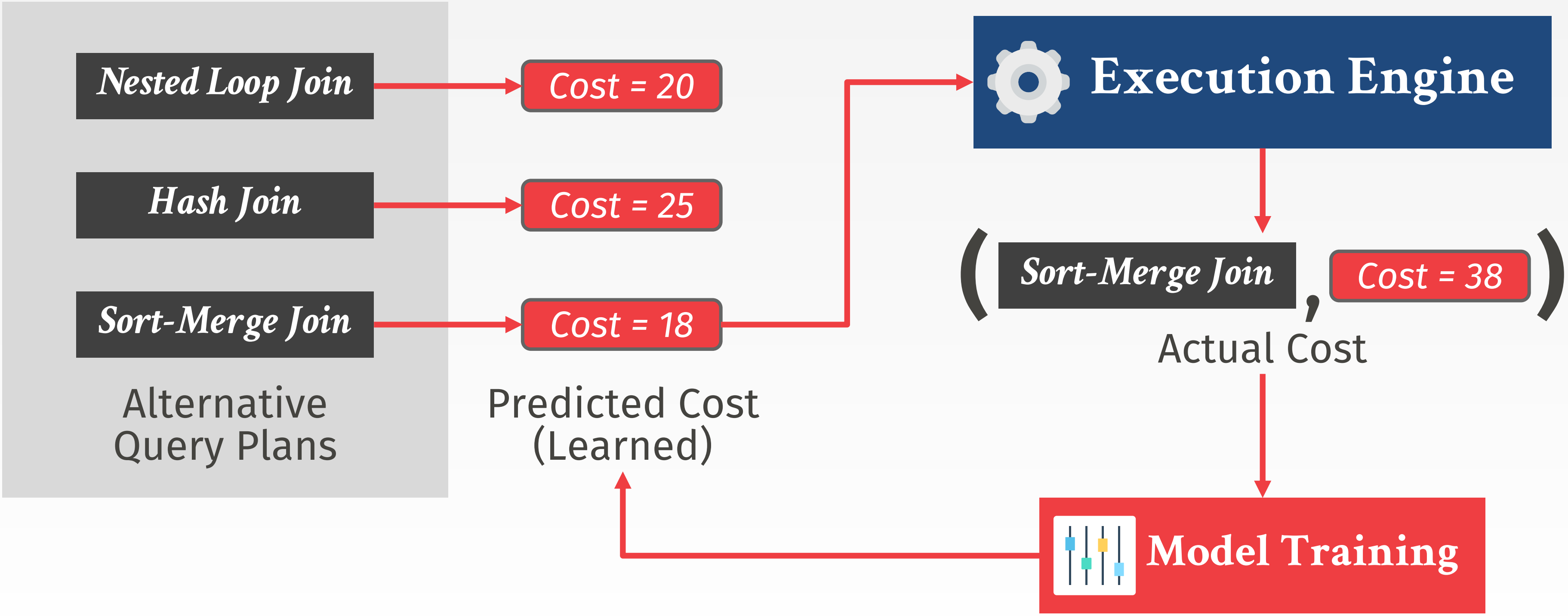
Configuration

- *Knob Tuning*
- *Partitioning*
- *Physical Design*

```
SELECT *
  FROM X JOIN Y
    ON X.id = Y.id;
```

Source: Ryan Marcus

Traditional Optimizer



Source: Ryan Marcus

LEARNED DATABASE COMPONENTS
QUERY OPTIMIZATION

Traditional Optimiz

Nested Loop Join

Hash Join

Sort-Merge Join

Alternative Query Plans

LEO – DB2's LEarning Optimizer

Michael Stillger[3*], Guy Lohman[1], Volker Markl[1], Mokhtar Kandil[2]

[1]IBM Almaden Research Center, 650 Harry Road, K55/B1, San Jose, CA, 95139, USA
[2]IBM Canada Ltd., 1150 Eglinton Ave. E., Toronto, ON M3C 1H7, Canada
[3*]Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404, USA

mstilger@siebel.com, {lohman, marklv}@almaden.ibm.com, mkandil@ca.ibm.com

Abstract

Most modern DBMS optimizers rely upon a cost model to choose the best query execution plan (*QEP*) for any given query. Cost estimates are heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the QEP for complex queries involving many predicates and/or operations. These estimates rely upon statistics on the database and modeling assumptions that may or may not be true for a given database. In this paper we introduce LEO, DB2's LEarning Optimizer, as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO compares the optimizer's estimates with actuals at each step in a QEP, and computes adjustments to cost estimates and statistics that may be used during future query optimizations. This analysis can be done either on-line or off-line on a separate system, and either incrementally or in batches. In this way, LEO introduces a feedback loop to query optimization that enhances the available information on the database where the most queries have occurred, allowing the optimizer to actually learn from its past mistakes. Our technique is general and can be applied to any operation in a QEP, including joins, derived results after several predicates have been applied, and even to DISTINCT and GROUP-BY operators. As shown by performance measurements on a 10 GB TPC-H data set, the runtime overhead of LEO's monitoring is insignificant, whereas the potential benefit to response time from more accurate cardinality and cost estimates can be orders of magnitude.

1. Introduction

Most modern query optimizers for relational database management systems (*DBMSs*) determine the best query execution plan (*QEP*) for executing an SQL query by mathematically modeling the execution cost for each plan and choosing the cheapest QEP. This execution cost is largely dependent upon the number of rows that will be processed by each operator in the QEP. Estimating the number of rows – or *cardinality* – after one or more predicates have been applied has been the subject of much research for over 20 years [SAC+79, Gel93, SS94, ARM89, Lyn88]. Typically this estimate relies on statistics of database characteristics, beginning with the number of rows for each table, multiplied by a *filter factor* – or *selectivity* – for each predicate, derived from the number of distinct values and other statistics on columns. The selectivity of a predicate P effectively represents the probability that any row in the database will satisfy P.

While query optimizers do a remarkably good job of estimating both the cost and the cardinality of most queries, many assumptions underlie this mathematical model. Examples of these assumptions include:

Currency of information: The statistics are assumed to reflect the current state of the database, i.e. that the database characteristics are relatively stable.

Uniformity: Although histograms deal with skew in values for "local" selection predicates (to a single table), we are unaware of any available product that exploits them for joins.

Independence of predicates: Selectivities for each predicate are calculated individually and multiplied together, even though the underlying columns may be related, e.g. by a functional dependency. While multi-dimensional histograms address this problem for local predicates, again they have never been applied to join predicates, aggregation, etc. Applications common today have hundreds of columns in each table and thousands of tables, making it impossible to know on which subset(s) of columns to maintain multi-dimensional histograms.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

* Work performed while the author was a post-doc at IBM ARC.

Bao: Making Learned Query Optimization Practical

Ryan Marcus, MIT & Intel Labs, ryanmarcus@csail.mit.edu
Parimarjan Negi, MIT, pnegi@csail.mit.edu
Hongzi Mao, MIT, hongzi@csail.mit.edu
Nesime Tatbul, MIT & Intel Labs, tatbul@csail.mit.edu
Mohammad Alizadeh, MIT, alizadeh@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization have shown few practical gains due to substantive training overhead, inability to adapt to changes, and poor tail performance. Motivated by these difficulties, we introduce Bao (the Bandit optimizer). Bao takes advantage of the wisdom built into existing query optimizers by providing per-query optimization hints. Bao combines modern tree convolutional neural networks with Thompson sampling, a well-studied reinforcement learning algorithm. As a result, Bao automatically learns from its mistakes and adapts to changes in query workloads, data, and schema. Experimentally, we demonstrate that Bao can quickly learn strategies that improve end-to-end query execution performance, including tail latency, for several workloads containing long-running queries. In cloud environments, we show that Bao can offer both reduced costs and better performance compared with a commercial system.

CS CONCEPTS

nformation systems → Query optimization.

EYWORDS

ery optimization; machine learning; reinforcement learning

M Reference Format:

n Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad deh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization tical. In *Proceedings of the 2021 International Conference on Management ata (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452838

INTRODUCTION

y optimization is an important task for database management ns. Despite decades of study [70], the most important elements ery optimization – cardinality estimation and cost modeling e proven difficult to crack [45]. Several works have applied ne learning techniques to these stubborn problems [37, 40, 44, 59, 72, 73, 76]. While all of these new solutions demonstrate kable results, we argue that none of the techniques are yet al, as they suffer from several fundamental problems:

(1) **Long training time.** Most proposed machine learning techniques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning require gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish to estimate cardinalities in the first place). Reinforcement learning techniques must process thousands of queries before outperforming traditional optimizers, which (when accounting for data collection and model training) can take on the order of days [51].

(2) **Inability to adjust to data and workload changes.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Cardinality estimators based on supervised learning must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques assume that both the workload and the schema remain constant, and require complete retraining when this is not the case [40, 51, 53, 59].

(3) **Tail catastrophe.** Recent work has shown that learning techniques can outperform traditional optimizers *on average*, but often perform catastrophically (e.g., 100x regression in query performance) in the tail [27, 51, 58, 60]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [76], such failures, even if rare, are unacceptable in many real world applications.

(4) **Black-box decisions.** While traditional cost-based optimizers are already complex, understanding query optimization is even harder when black-box deep learning approaches are used. Moreover, in contrast to traditional optimizers, current learned optimizers do not provide a way for database administrators to influence or understand the learned component's query planning.

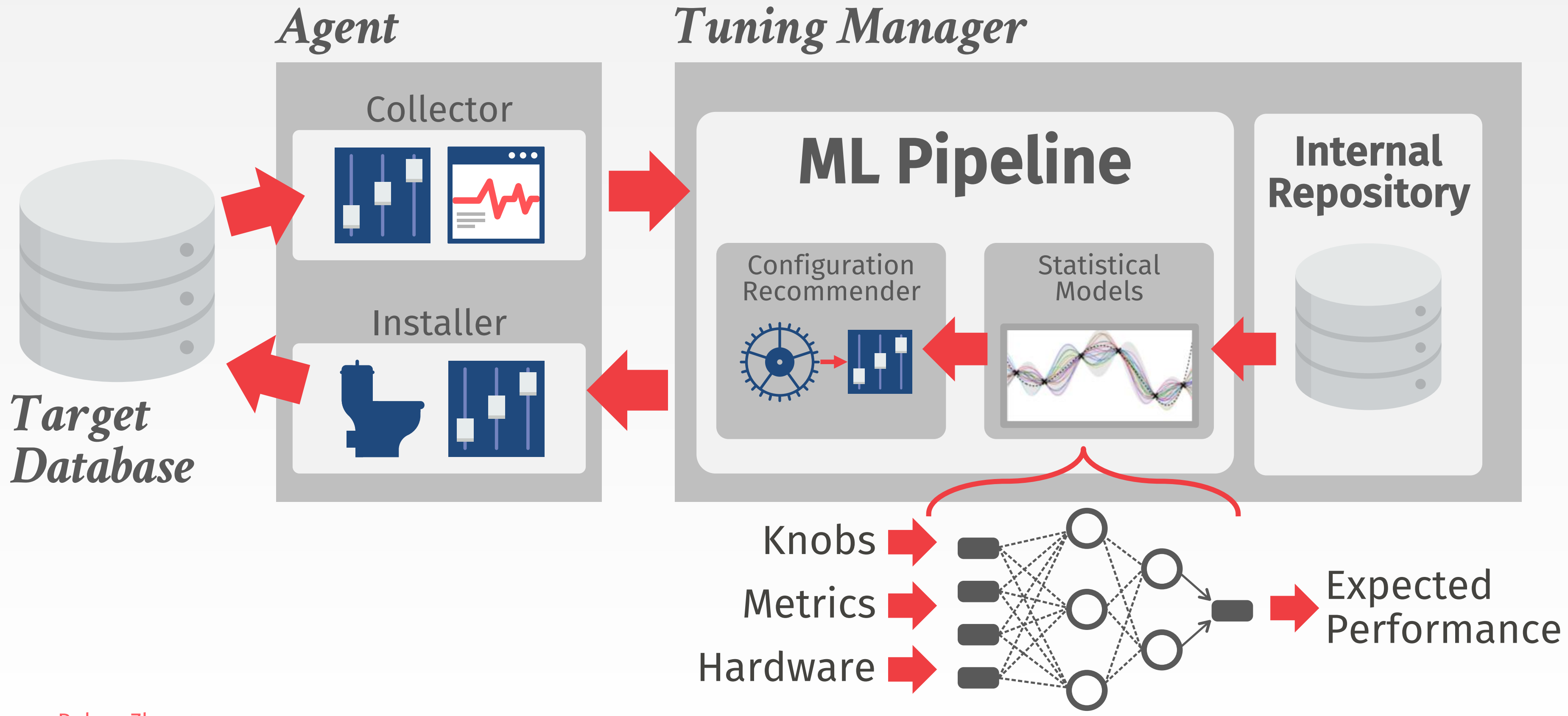
(5) **Integration cost.** To the best of our knowledge, all previous learned optimizers are still research prototypes, offering little to no integration with a real DBMS. None even supports all features of standard SQL, not to mention vendor specific features. Hence, fully integrating any learned optimizer into a commercial or open-source database system is not a trivial undertaking.

To the best of our knowledge, Bao (Bandit optimizer) is the first learned optimizer which overcomes the aforementioned problems. Bao is fully integrated into PostgreSQL as an extension, and can be easily installed without the need to recompile PostgreSQL. The database administrator (DBA) just needs to download our open-source module,[1] and even has the option to selectively turn the learned optimizer on or off for specific queries.

[1] https://learned.systems/bao

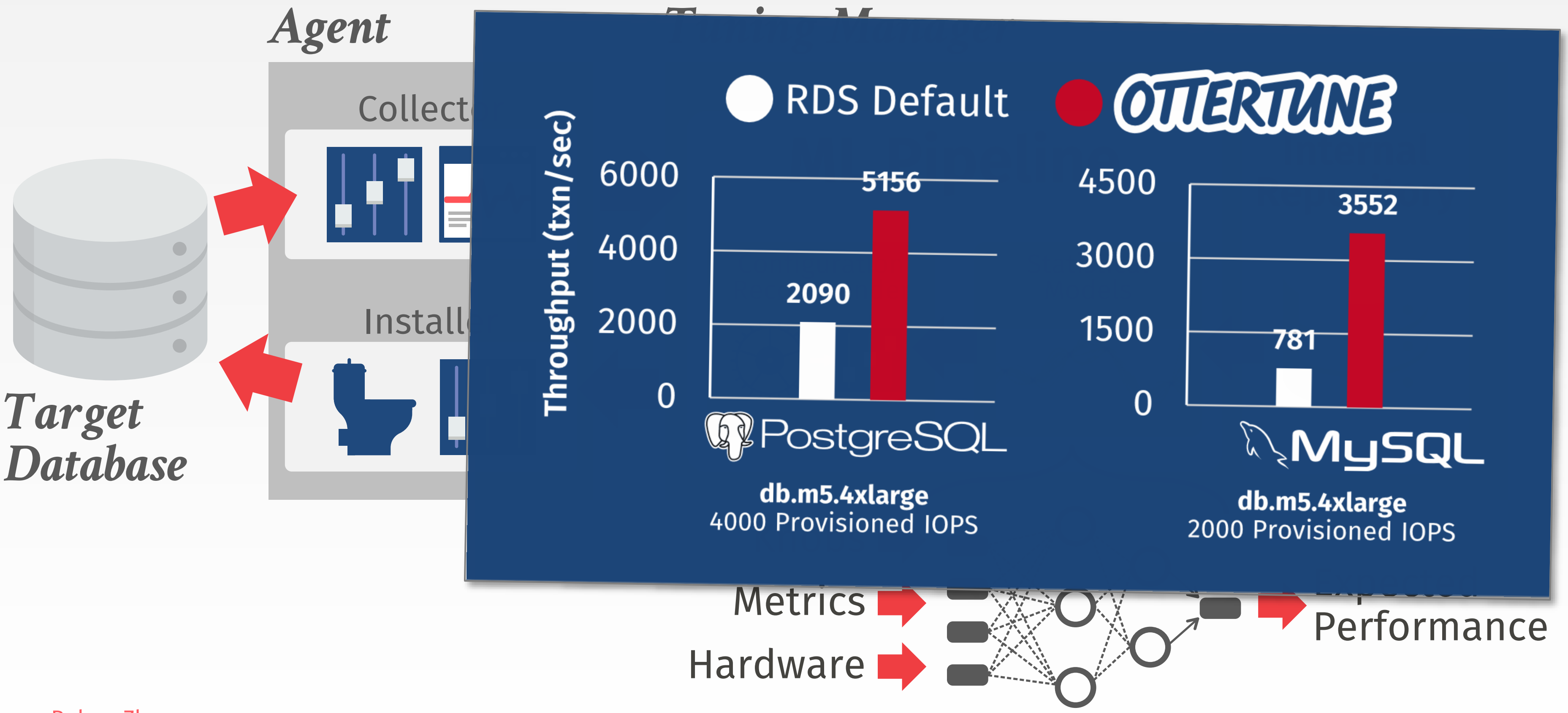
Source: Ryan Marcus

AUTOMATIC CONFIGURATION TUNING



Source: Bohan Zhang

LEARNED DATABASE COMPONENTS
AUTOMATIC CONFIGURATION TUNING



Source: Bohan Zhang

CHALLENGES

Failsafe Mechanisms?

Explainability?

Human Feedback / Overrides?

Transferability?

Does ML obviate the need for humans to build new database systems?

No.

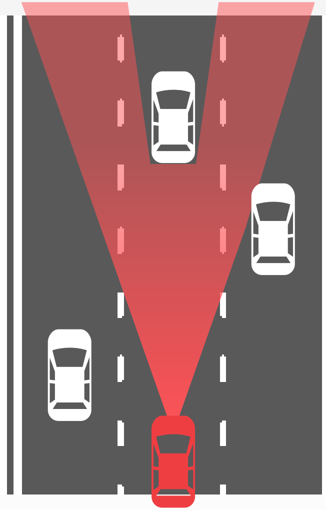
After we replace or supplement existing components with learned ones, what's next?

Challenge #1:

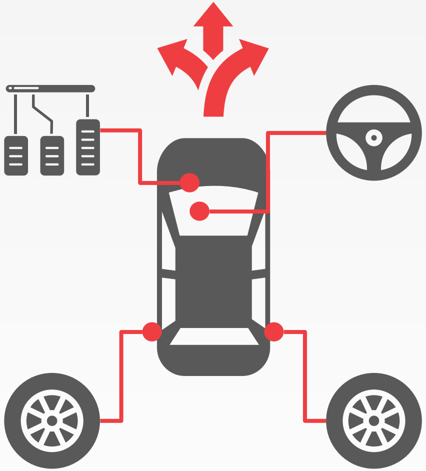
- *Remove the need for humans to perform any administrative task that does not require a human value judgement on externalities.*

Existing automation methods are reactive.
Humans are also proactive.

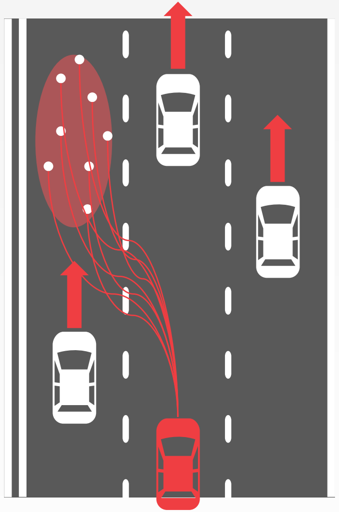
PERCEPTION



ACTION MODEL

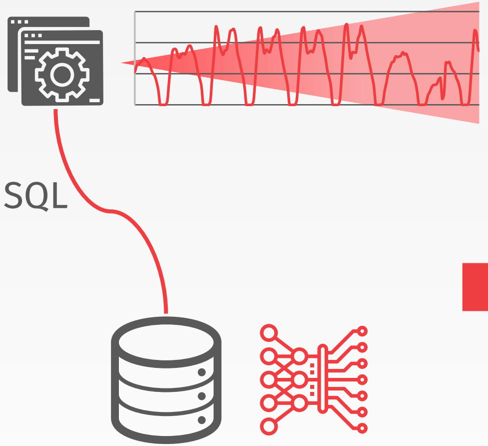


PLANNING



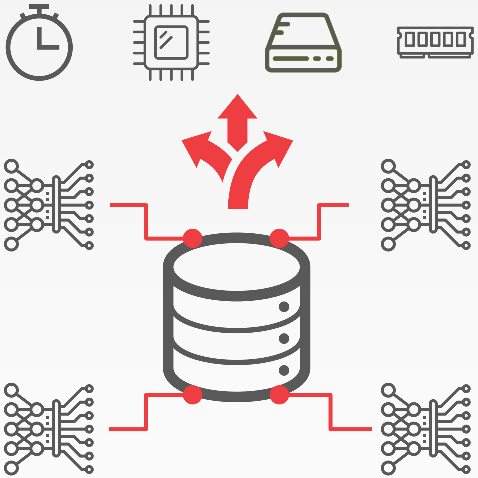
Source: Lin Ma

PERCEPTION



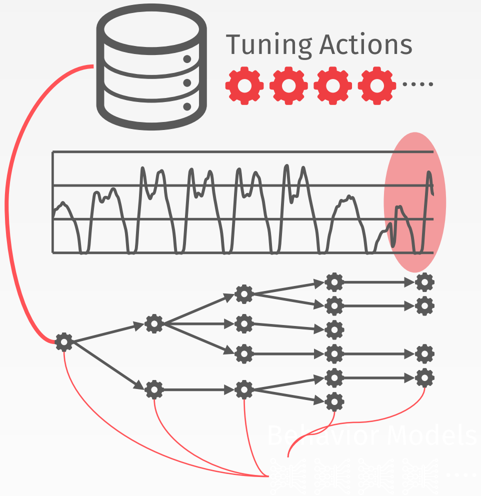
Workload Forecasting

ACTION MODEL



Behavior Modeling

PLANNING

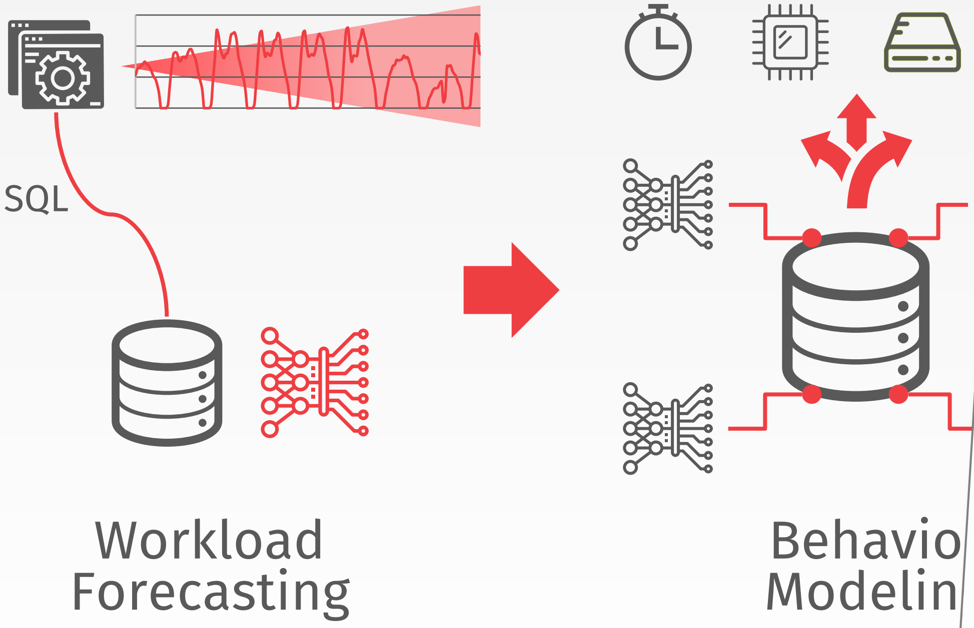


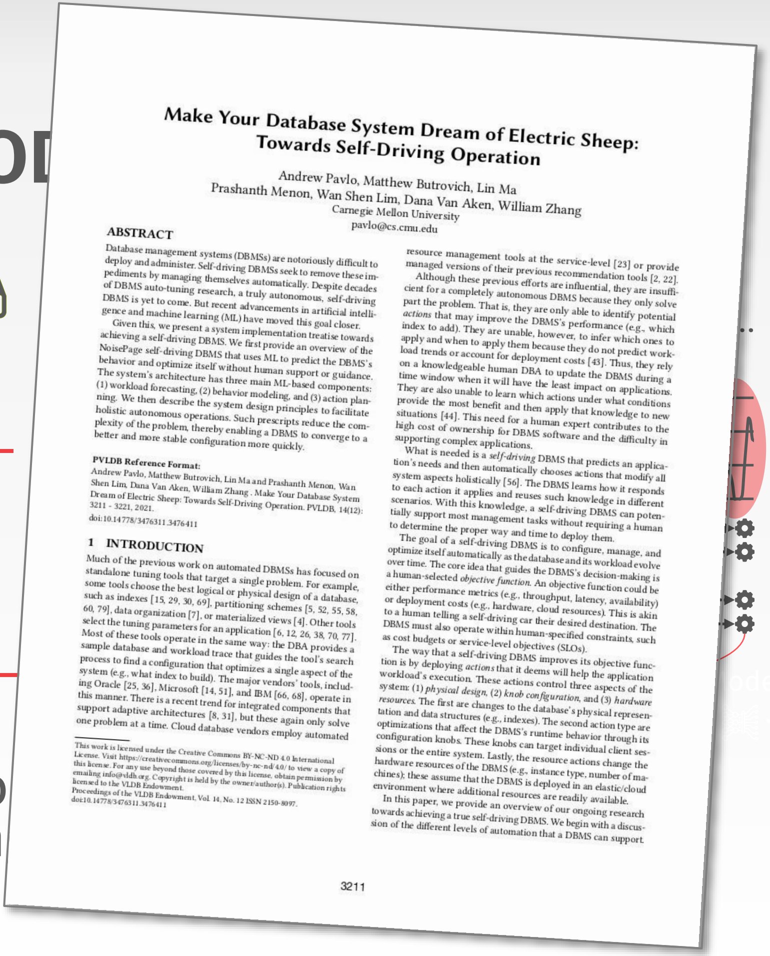
Action Planning

Source: Lin Ma

PERCEPTION

ACTION MOD





Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation

Andrew Pavlo, Matthew Butrovich, Lin Ma
Prashanth Menon, Wan Shen Lim, Dana Van Aken, William Zhang
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Database management systems (DBMSs) are notoriously difficult to deploy and administer. Self-driving DBMSs seek to remove these impediments by managing themselves automatically. Despite decades of DBMS auto-tuning research, a truly autonomous, self-driving DBMS is yet to come. But recent advancements in artificial intelligence and machine learning (ML) have moved this goal closer.

Given this, we present a system implementation treatise towards achieving a self-driving DBMS. We first provide an overview of the NoisePage self-driving DBMS that uses ML to predict the DBMS's behavior and optimize itself without human support or guidance. The system's architecture has three main ML-based components: (1) workload forecasting, (2) behavior modeling, and (3) action planning. We then describe the system design principles to facilitate holistic autonomous operations. Such prescripts reduce the complexity of the problem, thereby enabling a DBMS to converge to a better and more stable configuration more quickly.

PVLDB Reference Format:
Andrew Pavlo, Matthew Butrovich, Lin Ma and Prashanth Menon, Wan Shen Lim, Dana Van Aken, William Zhang . Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. PVLDB, 14(12): 3211 - 3221, 2021.
doi:10.14778/3476311.3476411

1 INTRODUCTION

Much of the previous work on automated DBMSs has focused on standalone tuning tools that target a single problem. For example, some tools choose the best logical or physical design of a database, such as indexes [15, 29, 30, 69], partitioning schemes [5, 52, 55, 58, 60, 79], data organization [7], or materialized views [4]. Other tools select the tuning parameters for an application [6, 12, 26, 38, 70, 77]. Most of these tools operate in the same way: the DBA provides a sample database and workload trace that guides the tool's search process to find a configuration that optimizes a single aspect of the system (e.g., what index to build). The major vendors' tools, including Oracle [25, 36], Microsoft [14, 51], and IBM [66, 68], operate in this manner. There is a recent trend for integrated components that support adaptive architectures [8, 31], but these again only solve one problem at a time. Cloud database vendors employ automated resource management tools at the service-level [23] or provide managed versions of their previous recommendation tools [2, 22].

Although these previous efforts are influential, they are insufficient for a completely autonomous DBMS because they only solve part the problem. That is, they are only able to identify potential *actions* that may improve the DBMS's performance (e.g., which index to add). They are unable, however, to infer which ones to apply and when to apply them because they do not predict workload trends or account for deployment costs [43]. Thus, they rely on a knowledgeable human DBA to update the DBMS during a time window when it will have the least impact on applications. They are also unable to learn which actions under what conditions provide the most benefit and then apply that knowledge to new situations [44]. This need for a human expert contributes to the high cost of ownership for DBMS software and the difficulty in supporting complex applications.

What is needed is a *self-driving* DBMS that predicts an application's needs and then automatically chooses actions that modify all system aspects holistically [56]. The DBMS learns how it responds to each action it applies and reuses such knowledge in different scenarios. With this knowledge, a self-driving DBMS can potentially support most management tasks without requiring a human to determine the proper way and time to deploy them.

The goal of a self-driving DBMS is to configure, manage, and optimize itself automatically as the database and its workload evolve over time. The core idea that guides the DBMS's decision-making is a human-selected *objective function*. An objective function could be either performance metrics (e.g., throughput, latency, availability) or deployment costs (e.g., hardware, cloud resources). This is akin to a human telling a self-driving car their desired destination. The DBMS must also operate within human-specified constraints, such as cost budgets or service-level objectives (SLOs).

The way that a self-driving DBMS improves its objective function is by deploying *actions* that it deems will help the application workload's execution. These actions control three aspects of the system: (1) *physical design*, (2) *knob configuration*, and (3) *hardware resources*. The first are changes to the database's physical representation and data structures (e.g., indexes). The second action type are optimizations that affect the DBMS's runtime behavior through its configuration knobs. These knobs can target individual client sessions or the entire system. Lastly, the resource actions change the hardware resources of the DBMS (e.g., instance type, number of machines); these assume that the DBMS is deployed in an elastic/cloud environment where additional resources are readily available.

In this paper, we provide an overview of our ongoing research towards achieving a true self-driving DBMS. We begin with a discussion of the different levels of automation that a DBMS can support.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476411

Source: Lin Ma

Challenge #2:

– *Discover new optimizations and techniques that are currently unknown to humans.*

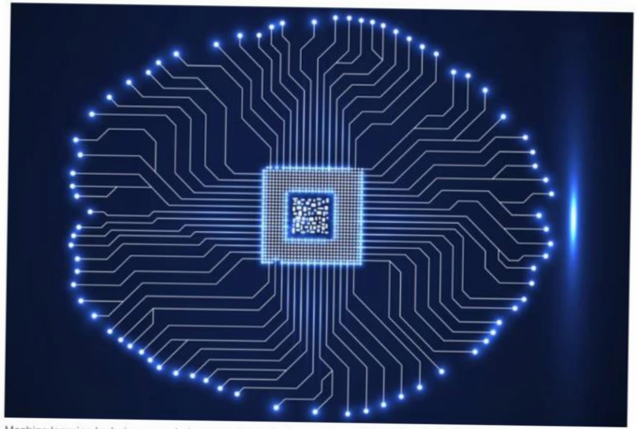
SCIENTIFIC AMERICAN.

ARTIFICIAL INTELLIGENCE

AI Generates Hypotheses Human Scientists Have Not Thought Of

Machine-learning algorithms can guide humans toward new experiments and theories

By Robin Blades on October 28, 2021



Machine learning techniques can help researchers develop novel hypotheses. Credit: Getty Images

Electric vehicles have the potential to substantially reduce carbon emissions, but car companies are running out of materials to make batteries. One crucial component, nickel, is projected to cause supply shortages as early as the end of this year. Scientists recently discovered four new materials that could potentially help—and what may be

Challenge #2:

– *Discover new optimizations and techniques that are currently unknown to humans.*

This requires a DBMS to have good introspection and instrumentation hooks/APIs.

There are less things to automatically optimize in an OLTP DBMS than in an OLAP DBMS.

– *There are fundamental limitations that prevent achieving even higher OLTP performance.*

Further methods will require automatically inferring higher-level semantics.

– *Example: Does an application really need all columns if it executes "*`SELECT *`*"?*

Current ML methods are trying to create better versions of existing DBMS components.

The next challenge is how to use ML to develop optimizations that humans would not think of on their own.

END

@andy_pavlo