DO WE STILL NEED PEOPLE TO WRITE DATABASE SYSTEMS?

OSACON 2021
#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 flourished 2000s because more organizations 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
Silicon Valley Sciences, Inc.
stonebraker@csail.mit.edu

Abstract

The last 30 years of commercial DBMS development can be summed upon a single phrase: “One size fits all.” This phrase refers to the fact that the traditional DBMS architecture, originally designed and structured for business data processing, has been used to support many data-intensive applications with varying storage characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database model, and that the commercial world will fracture into a collection of independent database engines, some of which may be used to a common frame-of-reference. We use examples from the stream-processing market and other data-warehousing market to illustrate our claims. We also briefly discuss other models for which the traditional architecture is a poor fit and argue for a critical rethinking of the current fact-warehousing systems running across industries.

1. Introduction

Traditional DBMSs served the needs of research prototypes in the 1970s in the form of System R [34] and INGRES [27]. The strong need of both prototypes was to support DBS in large customers or the applications that DBS was used for, namely “business data processing.” Thus, both DBMSs were identified for using transaction processing (TPP) applications, and their contingent counterparts (i.e., DDS and INGRES, respectively) found acceptance in this area in the 1980s. Other vendors (e.g., Sybase, Oracle, and Informix) followed the same path (DBMS code, which stores relational tables e.g. row-by-row, uses B-tree for indexing using a two-node storage, and provides ACID transactions properties).

Since the early 1990s, the major DBMS vendors have steadily sized to “one size fits all” strategy, whereby they maintain “one-size-mix-in” DBMS services. The reasons for this change 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 be against every code line,
- a size problem, because different code base is involved about which product to set as a customer, and
- a market penetration, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all major DBMS vendors have followed the advice “put all world behind one umbrella.” In this paper, we argue that this strategy has failed already, and will fail more dramatically in the future.

The rest of the paper is organized as follows. In Section 2, we broadly indicate why the single-code-line strategy has failed already by noting some of the key characteristics of the data-warehousing market. In Section 3, we discuss stream-processing applications and indicate a particular example where a specialized stream-processing engine is preferred by use of an ingredient. Sections 4 to 7 turn to the reasons for the performance difference, and indicate that DBMS technology is not likely to be able to adapt in the competitive data-warehousing market. In Section 7, we discuss the current state of other vendors where one size is likely to fit all, and other specialized database systems may be feasible.

2. Data warehousing

In the early 1990s, a new trend appeared — enterprises would gather together data from multiple operational systems into a data warehouse for business intelligence systems.
2000s

Columnar Storage

```
SELECT COUNT(B)
FROM XXX
WHERE A > ?;
```
2000s

Columnar Storage
Disaggregated Storage

Shared Nothing

Shared Disk
Vectorized Scan

```python
i = 0
for v_t in table:
    simdLoad(v_t.key, v_k)
    v_m = (v_k≥low ? 1 : 0) &
    (v_k≤high ? 1 : 0)
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

`SELECT * FROM table WHERE key >= "G" AND key <= "T"`
### Analytical Database Systems

**Last 20 Years**

**2000s**
- Columnar Storage
- Disaggregated Storage
- Vectorized Execution

#### SQL Query

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

#### Diagram

- **Key Vector**: `W U T A N G`
- **Mask**: `0 0 1 0 1 1`
- **All Offsets**: `0 1 2 3 4 5`
- **Matched Offsets**: `2 4 5`
**2010s**

**JIT Query Compilation**

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

*Expression Tree*
bool sel_eq_row(string str_col, string val0, int int_col, int val1) {
    return (str_col == val0 && int_col == val1);
}
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 great idea. I for one jumped on this bandwagon 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)
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 2014 ms for the system calls and 38738ms for the UDF. That’s 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(uho.padLeft(o.name + c.name, 100, '0'))
FROM msdb.sys.columns o
CROSS JOIN msdb.sys.columns c

Source: Karthik Ramachandra
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);

2010s
JIT Query Compilation
UDF Inlining

Froid: Optimization of Imperative Programs in a Relational Database

Karthik Ramachandra
Microsoft Gray Systems Lab
karthikr@ microsoft.com

Kwankhun Park
Microsoft Gray Systems Lab
kwpark@microsoft.com

Alban Hajarsson
Microsoft Gray Systems Lab
alan@microsoft.com

César Galindo-Leguizamón
Microsoft
microsoft.com

ABSTRACT
For decades, Relational databases have supported imperative SQL as a flexible extension to declarative SQL, allowing users to express queries procedurally and gain access to data from outside the database. However, this flexibility comes at a cost, as imperative SQL statements can lead to inefficient query execution plans and poor performance. In Froid, we propose a novel approach to optimizing imperative programs in a relational database. We use procedural code compilation to transform imperative SQL statements into efficient, optimized C code, which is then executed by the database engine. This approach allows Froid to automatically generate efficient code for executing complex imperative programs, leading to significant performance improvements over traditional imperative SQL approaches.

1. INTRODUCTION
SQL is popularly one of the most widely used programming languages in database systems. SQL is a declarative language for querying and manipulating data stored in relational databases. It allows users to express queries in a high-level, structured format, making it easy and efficient for querying large datasets.

However, SQL is limited in its ability to express complex, procedural code, such as iterative loops, conditionals, and recursive functions. These features are crucial for implementing complex algorithms and optimizing database operations. To overcome these limitations, imperative SQL (SQL with procedures) has been introduced in various database systems. However, this approach often leads to inefficient query execution plans and poor performance.

In Froid, we propose a novel approach to optimizing imperative programs in a relational database. We use procedural code compilation to transform imperative SQL statements into efficient, optimized C code, which is then executed by the database engine. This approach allows Froid to automatically generate efficient code for executing complex imperative programs, leading to significant performance improvements over traditional imperative SQL approaches.

We present Froid, an end-to-end framework for optimizing imperative programs in relational databases. Froid's approach leverages the power of modern compiler techniques to translate imperative SQL statements into efficient C code, which is then executed by the database engine. This approach allows Froid to automatically generate efficient code for executing complex imperative programs, leading to significant performance improvements over traditional imperative SQL approaches.

In Froid, we propose a novel approach to optimizing imperative programs in a relational database. We use procedural code compilation to transform imperative SQL statements into efficient, optimized C code, which is then executed by the database engine. This approach allows Froid to automatically generate efficient code for executing complex imperative programs, leading to significant performance improvements over traditional imperative SQL approaches.

XML-based queries can be used to optimize and manage large databases more efficiently. XML-based binary data structures are used to store and manage large datasets, while XML-based binary data structures are used to store and manage large datasets. XML-based binary data structures are used to store and manage large datasets.
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)
AS DT2
SELECT returnVal FROM
(SELECT 'high value' AS returnVal)
AS DT1
OUTER APPLY
(SELECT DT1.val + ' value' AS returnVal)
AS DT2
SELECT returnVal FROM
(SELECT 'high value' AS returnVal)
AS DT1
SELECT 'high value';

Source: Karthik Ramachandra
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)
AS DT2

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)
AS DT2

SELECT 'high value';

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

Dead Code Elimination
Const Propagation & Folding
Froid Inlining
Dynamic Slicing

Source: Karthik Ramachandra
Scalar UDF Inlining

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

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:

Source: Karthik Ramachandra
A learned component is an implemented portion of a DBMS that uses ML on previous observations to determine its future behavior as opposed to a human-devised strategy.
2020s
Learned Components

Traditional Index

Learned Index

Sorted Data

Sorted Data
LEARNED DATABASE COMPONENTS
RESEARCH EXAMPLES

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;
Traditional Optimizer

**Nested Loop Join**
- Cost = 20

**Hash Join**
- Cost = 25

**Sort-Merge Join**
- Cost = 18

Alternative Query Plans

Execution Engine
- **Sort-Merge Join**, Cost = 38

Predicted Cost (Learned)

Actual Cost

Model Training

Source: Ryan Marcus
Abstract

Recent advances in machine learning (ML) have led to the development of a new class of query optimization techniques that leverage ML models to predict the cost of query execution plans. These methods are particularly useful for data-intensive applications, such as data warehouses, where the cost of query execution can significantly impact performance.

In this paper, we present a novel approach to query optimization that uses ML models to predict the cost of different query execution plans. Our method involves training a ML model on a dataset of historical query execution data, and then using this model to predict the cost of different execution plans for new queries. We evaluate our approach on a variety of synthetic and real-world datasets, and show that it outperforms traditional query optimization techniques in terms of both predictive accuracy and execution time.

Introduction

Query optimization is a key component of database management systems (DBMSs). The goal of query optimization is to select the most efficient execution plan for a given query, which can significantly impact the performance of the database system.

Traditionally, query optimization is performed using cost-based optimization techniques. These techniques involve evaluating different execution plans and selecting the one with the lowest estimated cost. However, in recent years, machine learning (ML) has been applied to query optimization, with the aim of improving the accuracy and efficiency of the optimization process.

In this paper, we present a novel approach to query optimization that uses ML models to predict the cost of different execution plans. Our method involves training a ML model on a dataset of historical query execution data, and then using this model to predict the cost of different execution plans for new queries. We evaluate our approach on a variety of synthetic and real-world datasets, and show that it outperforms traditional query optimization techniques in terms of both predictive accuracy and execution time.
LEARNED DATABASE COMPONENTS
AUTOMATIC CONFIGURATION TUNING

Target Database

Agent

Collector

Installer

Tuning Manager

ML Pipeline

Internal Repository

Configuration Recommender

Statistical Models

Knobs

Metrics

Hardware

Expected Performance

Source: Bohan Zhang
LEARNED DATABASE COMPONENTS
AUTOMATIC CONFIGURATION TUNING

Target Database
Agent
Collector
Installer
Tuning Manager

Target Database

Source: Bohan Zhang
LEARNED DATABASE COMPONENTS
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**

SQL

Workload Forecasting

**ACTION MODEL**

Behavior Modeling

**PLANNING**

Tuning Actions

Action Planning

Source: Lin Ma
Challenge #2:

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