WHAT GOES AROUND COMES AROUND... AND AROUND...

Carnegie Mellon University

Andy Pavlo
Marcin Żukowski Coronation
CWI // November 2024

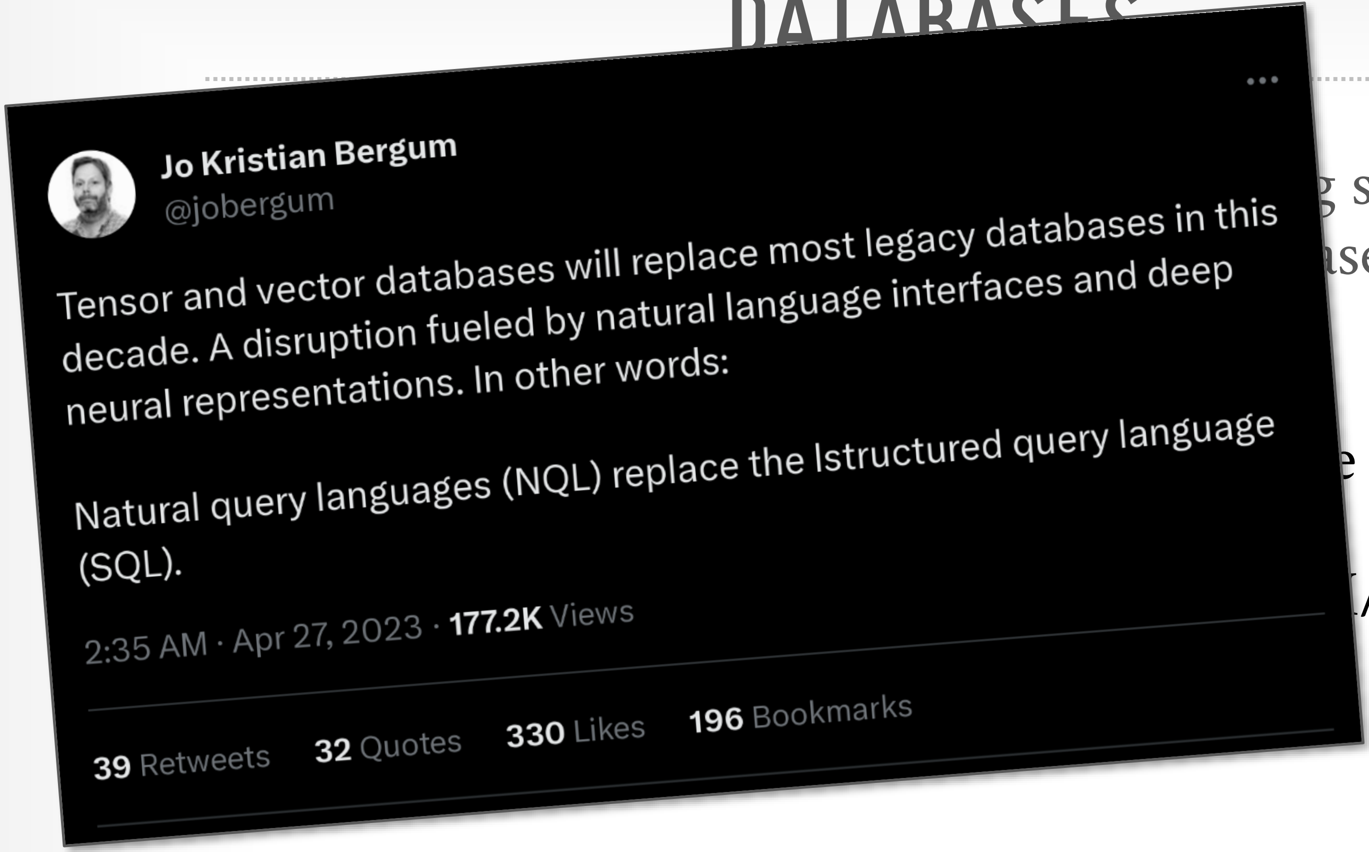
DATABASES

A database's **data model** is the underlying structure and organization of data within the database.

The **relational model** (RM) + **SQL** have dominated the database landscape since the 1980s.

But every 10 years somebody invents a RM/SQL "killer" that addresses some deficiency…

DATABASES

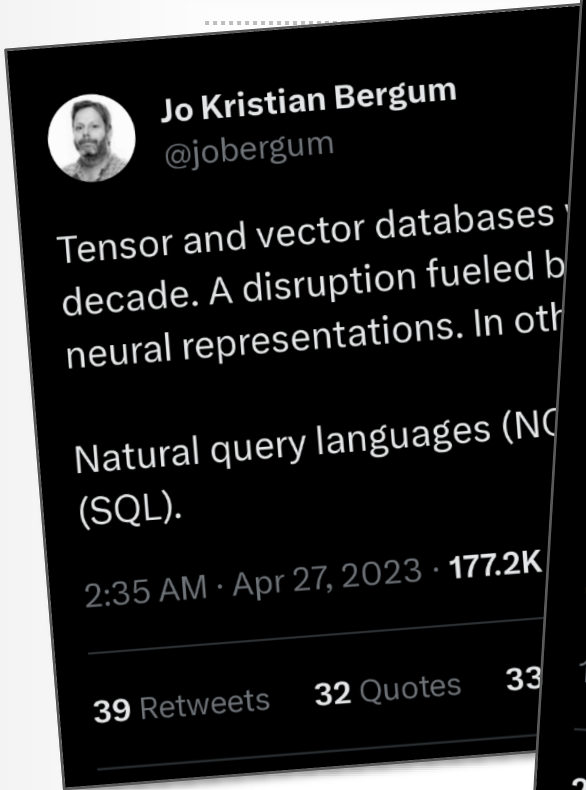


g structure
ase.

e 1980s.

/SQL

DATABASES


Jo Kristian Bergum
@jobergum

Tensor and vector databases
decade. A disruption fueled b
neural representations. In oth

Natural query languages (N(
(SQL).

2:35 AM · Apr 27, 2023 · 177.2K

39 Retweets 32 Quotes 33


Gagan Biyani 🏛
@gaganbiyani

SQL is going to die at the hands of an AI. I'm serious.

@mayowaoshin is already doing this. Takes your company's data and ingests it into ChatGPT. Then, you can create a chatbot for the data and just ask it questions using natural language.

This video demoes the output.

🤯

10:30 AM · May 18, 2023 · 2.6M Views

247 Retweets 203 Quotes 2,842 Likes 3,624 Bookmarks

SQL is **great**!

SQL is **bad**!

SQL is **great**!

NOT WEBSCALE
INCONSISTENT
AWKWARD
OLD
SLOW

SQL is **bad**!

SQL is **great**!

NOT WEBSCALE
INCONSISTENT
AWKWARD
OLD
SLOW

SQL is **bad**!

New Startups!
Lots of $$$!

SQL is **great**!

SQL is **bad**!

NOT WEBSCALE
INCONSISTENT
AWKWARD
OLD
SLOW

New Startups!
Lots of $$$!

SQL adopts new features

WHAT GOES AROUND COMES AROUND

READINGS IN DB SYSTEMS, 4TH EDITION (2005)

What Goes Around Comes Around

Michael Stonebraker
Joseph M. Hellerstein

Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Unfortunately, the main proposal in the current XML era bears a striking resemblance to the CODASYL proposal from the early 1970's, which failed because of its complexity. Hence, the current era is replaying history, and "what goes around comes around". Hopefully the next era will be smarter.

I Introduction

Data model proposals have been around since the late 1960's, when the first author "came on the scene". Proposals have continued with surprising regularity for the intervening 35 years. Moreover, many of the current day proposals have come from researchers too young to have learned from the discussion of earlier ones. Hence, the purpose of this paper is to summarize 35 years worth of "progress" and point out what should be learned from this lengthy exercise.

We present data model proposals in nine historical epochs:

Hierarchical (IMS): late 1960's and 1970's
Network (CODASYL): 1970's
Relational: 1970's and early 1980's
Entity-Relationship: 1970's
Extended Relational: 1980's
Semantic: late 1970's and 1980's
Object-oriented: late 1980's and early 1990's
Object-relational: late 1980's and early 1990's

https://cmudb.io/wga06

Hierarchical (1960s)

Network (1960s)

Relational (1970s)

Entity-Relationship (1970s)

Extended Relational (1980s)

Semantic (1980s)

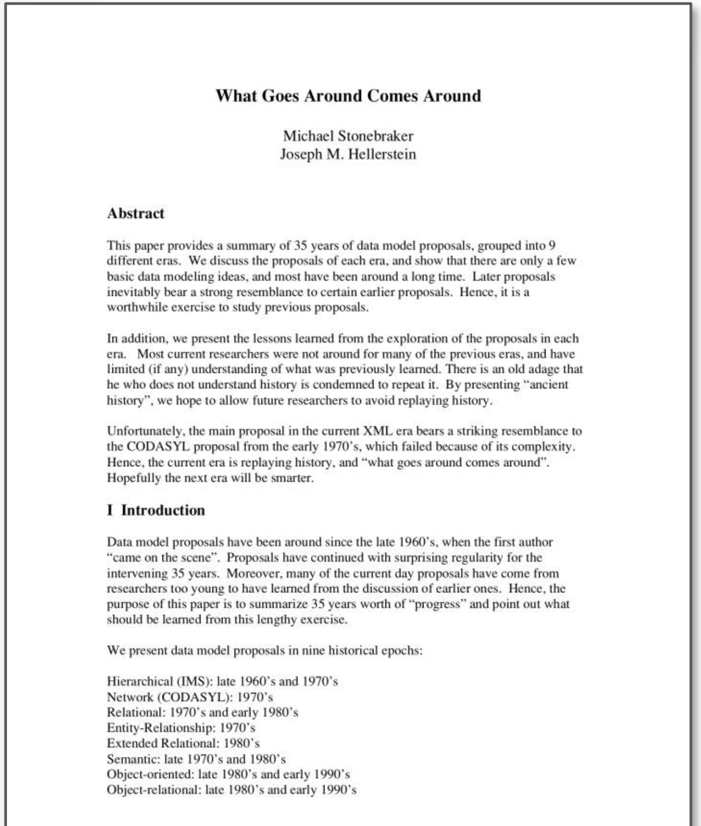
Object-Oriented (1980s)

Object-Relational (1990s)

Semi-Structured/XML (1990s)

WHAT GOES AROUND COMES AROUND

READINGS IN DB SYSTEMS, 4TH EDITION (2005)



What Goes Around Comes Around

Michael Stonebraker
Joseph M. Hellerstein

Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Unfortunately, the main proposal in the current XML era bears a striking resemblance to the CODASYL proposal from the early 1970's, which failed because of its complexity. Hence, the current era is replaying history, and "what goes around comes around". Hopefully the next era will be smarter.

I Introduction

Data model proposals have been around since the late 1960's, when the first author "came on the scene". Proposals have continued with surprising regularity for the intervening 35 years. Moreover, many of the current day proposals have come from researchers too young to have learned from the discussion of earlier ones. Hence, the purpose of this paper is to summarize 35 years worth of "progress" and point out what should be learned from this lengthy exercise.

We present data model proposals in nine historical epochs:

Hierarchical (IMS): late 1960's and 1970's
Network (CODASYL): 1970's
Relational: 1970's and early 1980's
Entity-Relationship: 1970's
Extended Relational: 1980's
Semantic: late 1970's and 1980's
Object-oriented: late 1980's and early 1990's
Object-relational: late 1980's and early 1990's

https://cmudb.io/wga06

Hierarchical (1960s)

Network (1960s) BCE

Relational (1970s)

Entity-Relationship (1970s)

Extended Relational (1980s)

Semantic (1980s)

Object-Oriented (1980s)

Object-Relational (1990s)

Semi-Structured/XML (1990s)

What Goes Around Comes Around

Michael Stonebraker
Joseph M. Hellerstein

Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Unfortunately, the main proposal in the current XML era bears a striking resemblance to the CODASYL proposal from the early 1970's, which failed because of its complexity. Hence, the current era is replaying history, and "what goes around comes around". Hopefully the next era will be smarter.

I Introduction

Data model proposals have been around since the late 1960's, when the first author "came on the scene". Proposals have continued with surprising regularity for the intervening 35 years. Moreover, many of the current day proposals have come from researchers too young to have learned from the discussion of earlier ones. Hence, the purpose of this paper is to summarize 35 years worth of "progress" and point out what should be learned from this lengthy exercise.

We present data model proposals in nine historical epochs:

Hierarchical (IMS): late 1960's and 1970's
Network (CODASYL): 1970's
Relational: 1970's and early 1980's
Entity-Relationship: 1970's
Extended Relational: 1980's
Semantic: late 1970's and 1980's
Object-oriented: late 1980's and early 1990's
Object-relational: late 1980's and early 1990's

https://cmudb.io/wga06

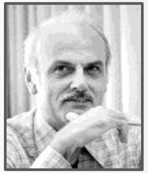
WHAT GOES AROUND COMES AROUND

READINGS IN DB SYSTEMS, 4TH EDITION (2005)

Hierarchical (1960s)

Network (1960s)

"Before Codd Era"



Relational (1970s)

Entity-Relationship (1970s)

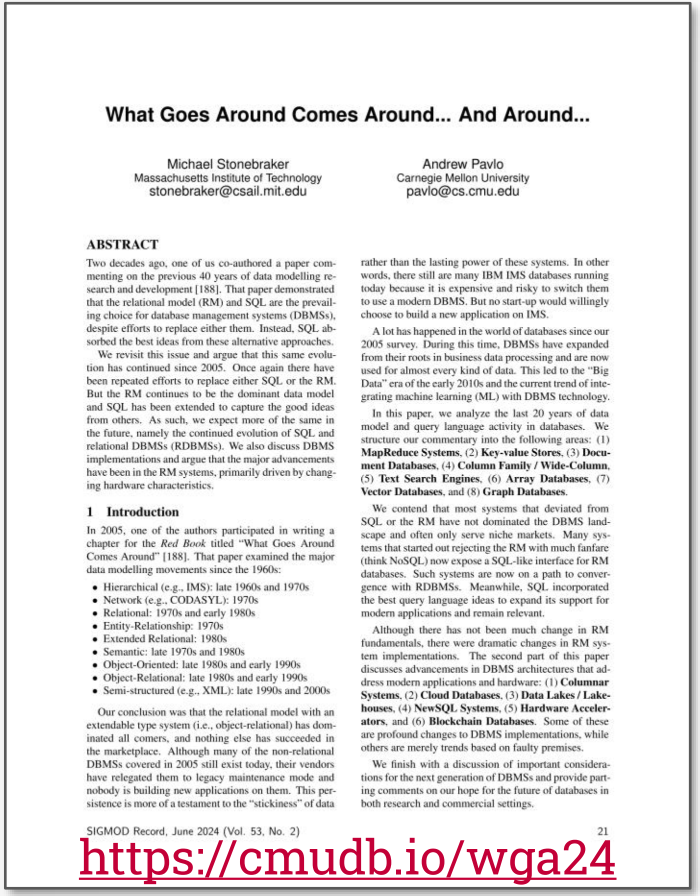
Extended Relational (1980s)

Semantic (1980s)

Object-Oriented (1980s)

Object-Relational (1990s)

Semi-Structured/XML (1990s)



What Goes Around Comes Around... And Around...

Michael Stonebraker
Massachusetts Institute of Technology
stonebraker@csail.mit.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Two decades ago, one of us co-authored a paper commenting on the previous 40 years of data modelling research and development [188]. That paper demonstrated that the relational model (RM) and SQL are the prevailing choice for database management systems (DBMSs), despite efforts to replace either them. Instead, SQL absorbed the best ideas from these alternative approaches.

We revisit this issue and argue that this same evolution has continued since 2005. Once again there have been repeated efforts to replace either SQL or the RM. But the RM continues to be the dominant data model and SQL has been extended to capture the good ideas from others. As such, we expect more of the same in the future, namely the continued evolution of SQL and relational DBMSs (RDBMSs). We also discuss DBMS implementations and argue that the major advancements have been in the RM systems, primarily driven by changing hardware characteristics.

1 Introduction

In 2005, one of the authors participated in writing a chapter for the *Red Book* titled "What Goes Around Comes Around" [188]. That paper examined the major data modelling movements since the 1960s:

- Hierarchical (e.g., IMS): late 1960s and 1970s
- Network (e.g., CODASYL): 1970s
- Relational: 1970s and early 1980s
- Entity-Relationship: 1970s
- Extended Relational: 1980s
- Semantic: late 1970s and 1980s
- Object-Oriented: late 1980s and early 1990s
- Object-Relational: late 1980s and early 1990s
- Semi-structured (e.g., XML): late 1990s and 2000s

Our conclusion was that the relational model with an extendable type system (i.e., object-relational) has dominated all comers, and nothing else has succeeded in the marketplace. Although many of the non-relational DBMSs covered in 2005 still exist today, their vendors have relegated them to legacy maintenance mode and nobody is building new applications on them. This persistence is more of a testament to the "stickiness" of data rather than the lasting power of these systems. In other words, there still are many IBM IMS databases running today because it is expensive and risky to switch them to use a modern DBMS. But no start-up would willingly choose to build a new application on IMS.

A lot has happened in the world of databases since our 2005 survey. During this time, DBMSs have expanded from their roots in business data processing and are now used for almost every kind of data. This led to the "Big Data" era of the early 2010s and the current trend of integrating machine learning (ML) with DBMS technology.

In this paper, we analyze the last 20 years of data model and query language activity in databases. We structure our commentary into the following areas: (1) **MapReduce Systems**, (2) **Key-value Stores**, (3) **Document Databases**, (4) **Column Family / Wide-Column**, (5) **Text Search Engines**, (6) **Array Databases**, (7) **Vector Databases**, and (8) **Graph Databases**.

We contend that most systems that deviated from SQL or the RM have not dominated the DBMS landscape and often only serve niche markets. Many systems that started out rejecting the RM with much fanfare (think NoSQL) now expose a SQL-like interface for RM databases. Such systems are now on a path to convergence with RDBMSs. Meanwhile, SQL incorporated the best query language ideas to expand its support for modern applications and remain relevant.

Although there has not been much change in RM fundamentals, there were dramatic changes in RM system implementations. The second part of this paper discusses advancements in DBMS architectures that address modern applications and hardware: (1) **Columnar Systems**, (2) **Cloud Databases**, (3) **Data Lakes / Lakehouses**, (4) **NewSQL Systems**, (5) **Hardware Accelerators**, and (6) **Blockchain Databases**. Some of these are profound changes to DBMS implementations, while others are merely trends based on faulty premises.

We finish with a discussion of important considerations for the next generation of DBMSs and provide parting comments on our hope for the future of databases in both research and commercial settings.

https://cmudb.io/wga24

WHAT GOES AROUND COMES AROUND... AND AROUND...

SIGMOD RECORD (2024)

- **Key-Value** (1990s)
- **MapReduce** (2000s)
- **Document/JSON** (2000s)
- **Column-family** (2000s)
- **Graph** (2000s)
- **Text Search** (1960s)
- **Array** (1990s)
- **Vector** (2020s)

Carnegie Mellon
Database Group

WHAT GOES AROUND COMES AROUND... AND AROUND...

SIGMOD RECORD (2024)


What Goes Around Comes Around... And Around...

Michael Stonebraker
Massachusetts Institute of Technology
stonebraker@csail.mit.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Two decades ago, one of us co-authored a paper commenting on the previous 40 years of data modelling research and development [188]. That paper demonstrated

Comes Around" [188]. That paper examined the major data modelling movements since the 1960s:

- Hierarchical (e.g., IMS): late 1960s and 1970s
- Network (e.g., CODASYL): 1970s
- Relational: 1970s and early 1980s
- Entity-Relationship: 1970s
- Extended Relational: 1980s
- Semantic: late 1970s and 1980s
- Object-Oriented: late 1980s and early 1990s
- Object-Relational: late 1980s and early 1990s
- Semi-structured (e.g., XML): late 1990s and 2000s

Our conclusion was that the relational model with an extendable type system (i.e., object-relational) has dominated all comers, and nothing else has succeeded in the marketplace. Although many of the non-relational DBMSs covered in 2005 still exist today, their vendors have relegated them to legacy maintenance mode and nobody is building new applications on them. This persistence is more of a testament to the "stickiness" of data rather than the lasting power of these systems. In other words, there still are many IBM IMS databases running today because it is expensive and risky to switch them

(think NoSQL) now expose a SQL-like interface for RM databases. Such systems are now on a path to convergence with RDBMSs. Meanwhile, SQL incorporated the best query language ideas to expand its support for modern applications and remain relevant.

Although there has not been much change in RM fundamentals, there were dramatic changes in RM system implementations. The second part of this paper discusses advancements in DBMS architectures that address modern applications and hardware: (1) **Columnar Systems**, (2) **Cloud Databases**, (3) **Data Lakes / Lakehouses**, (4) **NewSQL Systems**, (5) **Hardware Accelerators**, and (6) **Blockchain Databases**. Some of these are profound changes to DBMS implementations, while others are merely trends based on faulty premises.

We finish with a discussion of important considerations for the next generation of DBMSs and provide parting comments on our hope for the future of databases in both research and commercial settings.

TLDR: RM+SQL remains the best approach for most applications.

https://cmudb.io/wga24

Key-Value (1990s)

MapReduce (2000s)

Document/JSON (2000s)

Column-family (2000s)

Graph (2000s)

Text Search (1960s)

Array (1990s)

Vector (2020s)

KEY-VALUE STORES

Associative array that maps a key to a value.

→ Value is typically an untyped byte array that the DBMS cannot interpret.

```
(key, value)
```

Distributed KV Stores:

→ Shared-nothing DBMSs for caching + session data.

→ Provide higher/predictable performance instead of a more complex query language and features.

Embedded Storage Managers:

→ Low-level API systems that run in the same address space as a higher-level application.

KEY-VALUE STORES

Some distributed KV stores realized that expressive APIs are important and evolved into document stores.

→ If value is opaque, applications must implement more complex logic / types.

→ Better to start with a RM DBMS than to contort a KV DBMS to use a more complex data model (e.g., Postgres hstore).

Discussion:

→ Embedded KV storage managers make it easier to create full-featured DBMSs.

→ Very few commercial success stories for KV storage managers.

MAPREDUCE SYSTEMS

Distributed batch-oriented programming and execution model for analyzing large data sets.

Data model decided by user-written functions.
→ **Map**: UDF that performs computation + filtering
→ **Reduce**: Analogous to **GROUP BY** operation.

```
SELECT map() FROM crawl_table GROUP BY reduce();
```

MapReduce Frameworks:
→ Internal implementation at Google (2003).
→ Yahoo! created the open-source version Hadoop (2005).

DOI:10.1145/1629175.1629198

MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs.

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for processing and generating large data sets.[4] Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. We built a system around this programming model in 2003 to simplify construction of the inverted index for handling searches at Google.com. Since then, more than 10,000 distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, machine learning, and statistical machine translation. The Hadoop open source implementation of MapReduce has been used extensively outside of Google by a number of organizations.[10,11]

To help illustrate the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code like the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing required inter-machine communication. MapReduce allows programmers with no experience with parallel and distributed systems to easily utilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on hundreds or thousands of machines. Programmers find the system easy to use, and more than 100,000 MapReduce jobs are executed on Google's clusters every day.

Compared to Parallel Databases

The query languages built into parallel database systems are also used to

ILLUSTRATION BY MARIUS WATZ

UCE S

ed prog

data s

er-wri

omputa

UP BY

table

wor

on at

en-so

Carn

Data

DOI:10.1145/1629175.1629197

MapReduce complements DBMSs since databases are not designed for extract-transform-load tasks, a MapReduce specialty.

BY MICHAEL STONEBRAKER, DANIEL ABADI, DAVID J. DEWITT, SAM MADDEN, ERIK PAULSON, ANDREW PAVLO, AND ALEXANDER RASIN

MapReduce and Parallel DBMSs: Friends or Foes?

THE MAPREDUCE[7] (MR) PARADIGM has been hailed as a revolutionary new platform for large-scale, massively parallel data access.[16] Some proponents claim the extreme scalability of MR will relegate relational database management systems (DBMS) to the status of legacy technology. At least one enterprise, Facebook, has implemented a large data warehouse system using MR technology rather than a DBMS.[14]

Here, we argue that using MR systems to perform tasks that are best suited for DBMSs yields less than satisfactory results,[17] concluding that MR is more like an extract-transform-load (ETL) system than a DBMS, as it quickly loads and processes large amounts of data in an ad hoc manner. As such, it complements DBMS technology rather than competes with it. We also discuss the differences in the architectural decisions of MR systems and database systems and provide insight into how the systems should complement one another.

The technology press has been focusing on the revolution of "cloud computing," a paradigm that entails the harnessing of large numbers of processors working in parallel to solve computing problems. In effect, this suggests constructing a data center by lining up a large number of low-end servers, rather than deploying a smaller set of high-end servers. Along with this interest in clusters has come a proliferation of tools for programming them. MR is one such tool, an attractive option to many because it provides a simple model through which users are able to express relatively sophisticated distributed programs.

Given the interest in the MR model both commercially and academically, it is natural to ask whether MR systems should replace parallel database systems. Parallel DBMSs were first available commercially nearly two decades ago, and, today, systems (from about a dozen vendors) are available. As robust, high-performance computing platforms, they provide a high-level programming environment that is inherently parallelizable. Although it might seem that MR and parallel DBMSs are different, it is possible to write almost any parallel-processing task as either a set of database queries or a set of MR jobs.

Our discussions with MR users lead us to conclude that the most common use case for MR is more like an ETL system. As such, it is complementary to DBMSs, not a competing technology, since databases are not designed to be good at ETL tasks. Here, we describe what we believe is the ideal use of MR technology and highlight the different MR and parallel DMBS markets.

ILLUSTRATION BY MARIUS WATZ

MAPREDUCE SYSTEMS

People remembered that procedural query languages are (usually) a bad idea.

MR vendors put SQL engines on top of Hadoop.

Hadoop technology/services market crashed.

Google announced dropping MR in 2014.

Discussion:

→ Companies kept HDFS but replaced Hadoop compute layer with relational query engines.

→ Aspects of MR carried into distributed DBMSs (disaggregated compute/storage, shuffle phase).

DOCUMENT DATABASES

Represent a database as a collection of document objects that contain a hierarchy of field/value pairs.

→ Each document field is identified by a name.

→ A field's value is either a scalar type, array of values, or another document.

→ Applications do not predefine schema.

```
{<field>: <scalar|[values]|{document}>}
```

MongoDB

CouchDB

RAVENDB

Couchbase

NoSQL Document-oriented Systems:

→ Non-standard / procedural query languages

→ Defined by what they lack instead of what they provide.

DOCUMENT DATABASES

Document model is the same as previous models with many of the same problems.

→ **Object-Oriented** (1980s)

→ **Semi-Structured / XML** (1990s).

Core idea is denormalization ("pre-joining"):

→ Avoid object-relational impedance mismatch between application code and DBMS data model.

→ Avoid need for joins / multiple queries to retrieve data related to an object (N+1 SELECT Problem).

VERSANT

ObjectStore

O2 Technology

MarkLogic

DOCUMENT DATABASES

Almost every major NoSQL DBMS relearned (most) of the lessons from the 1970s:

→ SQL APIs are a good idea.
→ Schemas + integrity constraints are a good idea.
→ Transactions are a good idea.
→ Logical/physical data independence is a good idea.

DOCUMENT DATABASES

Almost every major NoSQL DBMS relearned (most) of the lessons from the 1970s:

→ SQL APIs are a good idea.
→ Schemas + integrity constraints are a good idea.
→ Transactions are a good idea.
→ Logical/physical data independence is a good idea.

amazon DynamoDB → PartiQL
cassandra → CQL
AEROSPIKE → AQL
Couchbase → SQL++

DOCUMEN

Almost every major No
(most) of the lessons fr
→ SQL APIs are a good ide
→ Schemas + integrity con
→ Transactions are a good
→ Logical/physical data in

MongoDB.

Introducing the Atlas SQL Interface, Connectors, and Drivers

Alexi Antonino
June 7, 2022 | Updated: June 8, 2022
#MongoDB World

We're excited to announce the Atlas SQL Interface, Connectors, and Drivers, which are now available for public preview. This feature empowers data analysts, many of whom are accustomed to working with SQL, to query and analyze Atlas data using their existing knowledge and preferred tools. Additionally, because the Atlas SQL Interface leverages Atlas Data Federation for its query engine, you can access data across Atlas clusters and cloud object stores using a single SQL query.

The Atlas SQL Connectors and Drivers allow you to connect MongoDB as a data source for your SQL-based business intelligence (BI) and analytics tools, resulting in faster insights and consistent analysis on the freshest data. You'll be able to seamlessly create visualizations and dashboards to more easily extract hidden value in your multi-structured data – without relying on time-consuming procedures like data movement or

DOCUMENT DATABASES

Almost every major NoSQL DBMS relearned (most) of the lessons from the 1970s:

→ SQL APIs are a good idea.
→ Schemas + integrity constraints are a good idea.
→ Transactions are a good idea.
→ Logical/physical data independence is a good idea.

Amazon DynamoDB → PartiQL
Cassandra → CQL
Aerospike → AQL
Couchbase → SQL++

Discussion:

→ SQL:2016 added JSON operators, SQL:2023 added JSON types.
→ The intellectual distance between relational+JSON DBMSs and document+SQL DBMSs has shrunk.

COLUMN-FAMILY / WIDE-COLUMN

Reduction of the document data model that only supports one level of nesting.

→ A record's value can only be a scalar or an array of scalars.

→ Deficiencies are the same as the document model.

```
{<field>: <scalar|[values]>}
```

Column-Family Systems:

→ First implementation was Google's Bigtable (2004)

→ Copied by several Internet start-ups.

COLUMN-FAMILY / WIDE-COLUMN

Reduction of the docu
supports one level of n

→ A record's value can on
→ Deficiencies are the san

```
{<field>: <scala
```

Column-Famil

→ First implement
→ Copied by sever

Google Cloud

Databases

Bigtable transforms the developer experience with SQL support

August 2, 2024

Christopher Crosbie
Group Product Manager, Google

Gary Elliott
Engineering Manager, Bigtable

Bigtable is a fast, flexible, NoSQL database that powers core Google services such as Search, Ads, and YouTube, as well as critical applications for customers such as PLAID and Mercari. Today, we're announcing Bigtable support for GoogleSQL, an ANSI-compliant SQL dialect used by Google products such as Spanner and BigQuery. Now you can use the same SQL with Bigtable to write applications for AI, fraud detection, data mesh, recommendations, or any other application that would benefit from real-time data.

Bigtable SQL support allows you to query Bigtable data using the familiar GoogleSQL syntax, making it easier for development teams to work with Bigtable's flexibility and speed. With over 100 SQL functions at launch, Bigtable SQL support also makes it easy to analyze and process large amounts of data directly within Bigtable, unlocking its potential for a wider range of use cases, ranging from JSON manipulation for log analysis, hyperloglog for web analytics, or kNN for vector search and generative AI.

GRAPH DATABASES

Direct multigraph structure that supports key/value labels for nodes and edges.

→ Property Graph vs. Resource Description Framework (RDF)

```
Node (id, {key: value}*)
Edge (node_id₁, node_id₂, {key: value}*)
```

neo4j

OrientDB

TigerGraph

MEM GRAPH

Property Graph DBMSs:

→ Provide graph-oriented traversal APIs.

→ Inefficient schemaless storage.

GRAPH DATABASES

Graph model is the same as the **network model** from CODASYL (1970s) with same issues.

Advancements in algorithms and systems will diminish the perceived advantage of specialized graph DBMSs.
→ Worst-case Optimal Joins
→ Vectorized Query Execution
→ Factorized Query Processing

Discussion:
→ SQL:2023 introduced SQL/PGQ (based on Neo4j Cypher, Oracle PGQL, TigerGraph GSQL) + emerging GQL standard.
→ Studies show that RM DBMSs outperform graph DBMSs.

GRAPH DATA

Graph model is the same as the n
CODASYL (1970s) with same iss

Advancements in algorithms and
the perceived advantage of speci

→ Worst-case Optimal Joins
→ Vectorized Query Execution
→ Factorized Query Processing

Discussion:

→ SQL:2023 introduced SQL/PGQ (
Oracle PGQL, TigerGraph GSQL)
→ Studies show that RM DBMSs out

DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS

Daniel ten Wolde, CWI, The Netherlands, dljtw@cwi.nl
Tavneet Singh, CWI, The Netherlands, tavneet.singh@cwi.nl
Gábor Szárnyas, CWI, The Netherlands, gabor.szarnyas@cwi.nl
Peter Boncz, CWI, The Netherlands, boncz@cwi.nl

ABSTRACT

In the past decade, property graph databases have emerged as a growing niche in data management. Many native graph systems and query languages have been created, but the functionality and performance still leave much room for improvement. The upcoming SQL:2023 will introduce the Property Graph Queries (SQL/PGQ) sub-language, giving relational systems the opportunity to standardize graph queries, and provide mature graph query functionality.

We argue that (i) competent graph data systems must build on all technology that makes up a state-of-the-art relational system, (ii) the graph use case requires the addition to that of a many-source/destination path-finding algorithm and compact graph representation, and (iii) incites research in practical worst-case-optimal joins and factorized query processing techniques.

We outline our design of DuckPGQ that follows this recipe, by adding efficient SQL/PGQ support to the popular open-source "embeddable analytics" relational database system DuckDB, also originally developed at CWI. Our design aims at minimizing technical debt using an approach that relies on efficient vectorized UDFs. We benchmark DuckPGQ showing encouraging performance and scalability on large graph data sets, but also reinforcing the need for future research under (iii).

1 INTRODUCTION

Graph Database systems have emerged as a growing niche in data management, with many property graph systems [7] such as Neo4j, TigerGraph, Dgraph, Titan and AWS Neptune becoming available, all using different query languages (i.e., Cypher, GSQL, GraphQL, Gremlin, SPARQL [2]). Property Graphs are directed graphs consisting of vertex and edge elements; where elements may have labels and associated key/value properties. Property graph systems are quite young, and performance of analytical queries on large graphs has been observed to be significantly lower than relational database systems, on graph queries that can also be formulated as SQL [16].

In RDBMS designs, there have been significant performance improvements in the past decade, with analytical systems such as Snowflake and Databricks adopting principles like skippable columnar storage with lightweight compression [24] (also popular in open-source formats such as Parquet and ORC), efficient load-balanced multi-core parallelism using "morsel-driven" scheduling [15] and efficient query execution techniques [14]: either using vectorized query execution or Just-In-Time low-level compilation of queries into executable programs.

The upcoming SQL:2023 introduces the **SQL/PGQ** (Property Graph Queries) sub-language [8], which allows (1) to define graph views over relational tables and (2) to formulate graph pattern matching and path-finding operations using a SQL syntax. These features narrow the functionality gap between RDBMSs and native graph systems, and unify the feature space with a common graph query sub-language, as PGQ is also a subset of the upcoming ISO Graph Query Language **GQL** [8] that native graph systems intend to adopt. GQL will add graph updates, querying multiple graphs and queries that return a *graph result*, rather than a binding *table*.

SQL/PGQ by example. If we have relational tables Student and College and connecting tables know and enrol, we can define a property graph pg consisting of Person vertexes connected to each other by edges with label know and to College vertexes via studiesAt edges:[1]

```
CREATE PROPERTY GRAPH pg
VERTEX TABLES(
  Student PROPERTIES(id,name,birthDate)    LABEL Person,
  College PROPERTIES(id,college))
EDGE TABLES(
  know  SOURCE Person KEY(id) DESTINATION Person KEY(id)
        PROPERTIES(createDate,msgCount),
  enrol SOURCE Student KEY(id) DESTINATION College KEY(id)
        PROPERTIES(classYear)               LABEL studiesAt)
```

In the below SELECT query the MATCH will bind variable a to all vertexes that satisfy a label-test :Person and have property name='Ana'. The comma separating the two pattern expressions implies a conjunction[2] with matching variable bindings: it requires a to also have an edge labeled studiesAt towards a College c:

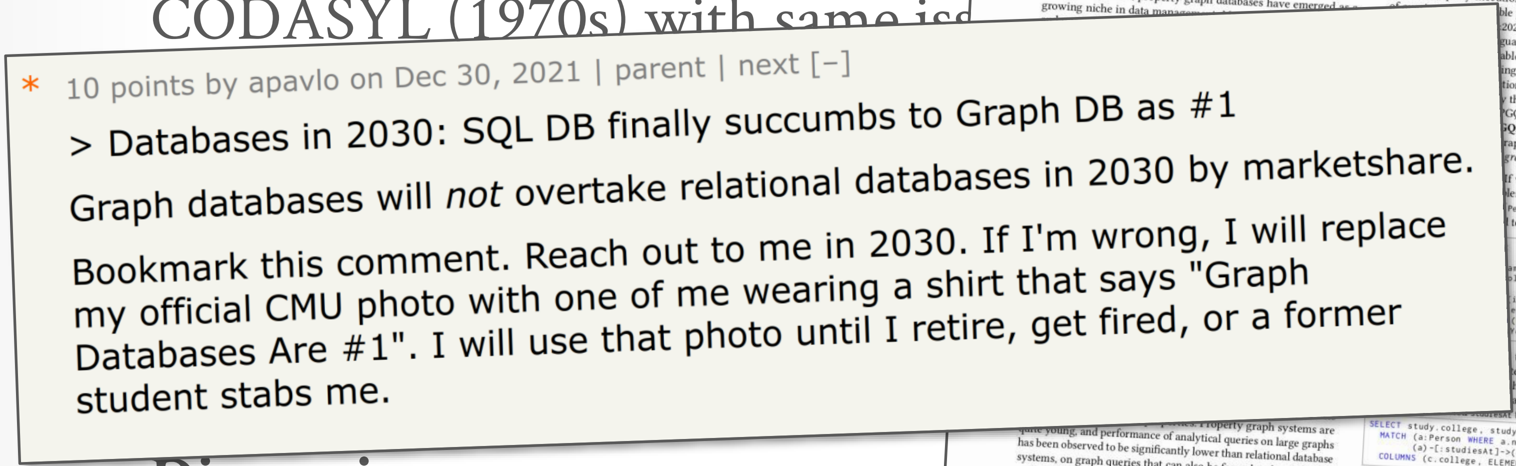
```
SELECT study.college, study.pid FROM GRAPH_TABLE (pg,
  MATCH (a:Person WHERE a.name='Ana'),
        (a)-[:studiesAt]->(c:College)
  COLUMNS (c.college, ELEMENT_ID(a) AS pid)) study
```

The MATCH clause produces a conceptual *binding table* with each row holding matched bindings and one column for each variable. These bindings denote elements (e.g., a vertex or edge); the COLUMNS clause retrieves scalar values from those. The example retrieves the property c.college and the implicit element identifier[3] of a, as the columns of a temporary GRAPH_TABLE named study in the FROM clause.

[1] The table name is the default label. DuckPGQ allows an additional LABEL list of max length 64, and a BIGINT LABEL FROM col *specifier* column. Elements only have a label from the list if their corresponding bit is set. This allows e.g., to express class membership with inheritance in labels. DuckPGQ will not support having the same label in multiple tables, as element patterns must always bind to a single table.
[2] Inside path expressions, the | will UNION pattern bindings, and |+| stands for UNION ALL; though neither is supported initially in DuckPGQ.
[3] ELEMENT_ID() is implementation-dependent; in DuckPGQ it returns a rowid.

GRAPH DATA

Graph model is the same as the n
CODASYL (1970s) with same iss


* 10 points by apavlo on Dec 30, 2021 | parent | next [–]

> Databases in 2030: SQL DB finally succumbs to Graph DB as #1

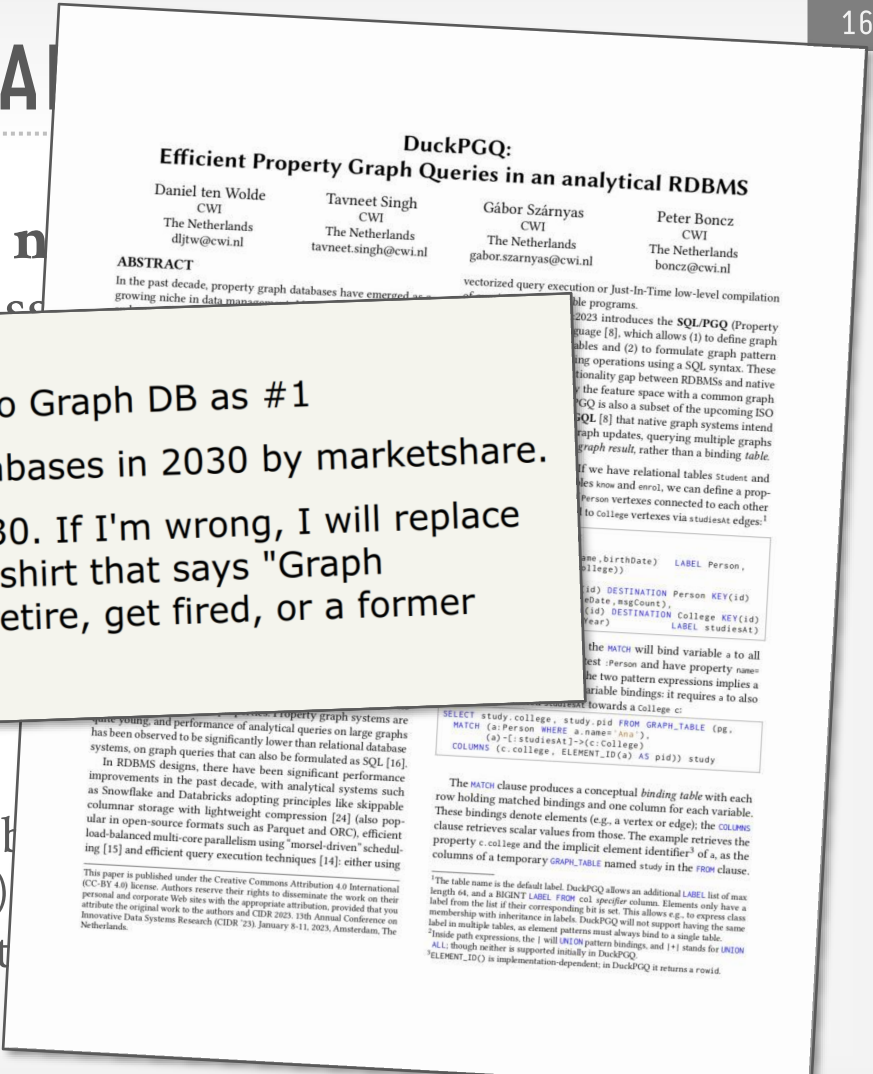
Graph databases will *not* overtake relational databases in 2030 by marketshare.

Bookmark this comment. Reach out to me in 2030. If I'm wrong, I will replace my official CMU photo with one of me wearing a shirt that says "Graph Databases Are #1". I will use that photo until I retire, get fired, or a former student stabs me.

Discussion:

→ SQL:2023 introduced SQL/PGQ (
Oracle PGQL, TigerGraph GSQL)

→ Studies show that RM DBMSs out


DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS

Daniel ten Wolde, CWI, The Netherlands, dljtw@cwi.nl
Tavneet Singh, CWI, The Netherlands, tavneet.singh@cwi.nl
Gábor Szárnyas, CWI, The Netherlands, gabor.szarnyas@cwi.nl
Peter Boncz, CWI, The Netherlands, boncz@cwi.nl

ABSTRACT

In the past decade, property graph databases have emerged as a growing niche in data management

quite young, and performance of analytical queries on large graphs has been observed to be significantly lower than relational database systems, on graph queries that can also be formulated as SQL [16].

In RDBMS designs, there have been significant performance improvements in the past decade, with analytical systems such as Snowflake and Databricks adopting principles like skippable columnar storage with lightweight compression [24] (also popular in open-source formats such as Parquet and ORC), efficient load-balanced multi-core parallelism using "morsel-driven" scheduling [15] and efficient query execution techniques [14]: either using vectorized query execution or Just-In-Time low-level compilation of queries into executable programs.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2023. 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

```
SELECT study.college, study.pid FROM GRAPH_TABLE (pg,
  MATCH (a:Person WHERE a.name='Ana'),
    (a)-[:studiesAt]->(c:College)
  COLUMNS (c.college, ELEMENT_ID(a) AS pid)) study
```

The MATCH clause produces a conceptual *binding table* with each row holding matched bindings and one column for each variable. These bindings denote elements (e.g., a vertex or edge); the COLUMNS clause retrieves scalar values from those. The example retrieves the property c.college and the implicit element identifier[3] of a, as the columns of a temporary GRAPH_TABLE named study in the FROM clause.

[1] The table name is the default label. DuckPGQ allows an additional LABEL list of max length 64, and a BIGINT LABEL FROM col *specifier* column. Elements only have a label from the list if their corresponding bit is set. This allows e.g., to express class membership with inheritance in labels. DuckPGQ will not support having the same label in multiple tables, as element patterns must always bind to a single table.
[2] Inside path expressions, the | will UNION pattern bindings, and |+| stands for UNION ALL; though neither is supported initially in DuckPGQ.
[3] ELEMENT_ID() is implementation-dependent; in DuckPGQ it returns a rowid.

TEXT SEARCH ENGINES

Systems that extract structure (e.g., meta-data, indexes) from text data and support queries over that content.

→ Tokenize documents into "bag of words" and then build inverted indexes over those tokens.

→ No data model because text data is inherently unstructured.

Core ideas pioneered by Cornell's SMART (1965).

elasticsearch OpenSearch splunk> APACHE LUCENE vespa QUICKWIT Solr tantivy

Text Search Engines:

→ Quickly parse, index, and store large documents.

→ Built-in support for noise/salient words + synonyms.

TEXT SEARCH ENGINES

Leading RM DBMSs include full-text search indexes but their adoption is stymied by non-model reasons.

→ Non-standard SQL operations / syntax.

→ Text data is large but not high importance. DBMS storage is always more expensive than generic storage.

Discussion:

→ Maintaining a separate text search DBMS should be unnecessary but lots of people still do it.

→ All DBMS vendors are augmenting inverted-index text search with vector-based similarity search…

ARRAY DATABASES

Collection of data where each element is identifiable by one or more dimension offsets.

→ Vectors (1D), Matrices (2D), Tensors (+3D)

→ Dimensions do not have to align with integer grids.

$(\texttt{dimension}_1, \texttt{dimension}_2, \ldots \texttt{[values]})$

Array DBMSs:

→ Specialized storage managers and execution engines.

→ Sparse vs. Dense Arrays

ARRAY DATABASES

Supporting arrays as first-class data types violates the original RM vision. But this is a good example of RM evolving to meet the needs of applications.

Discussion:

→ SQL:2023 added multi-dimensional arrays (SQL/MDA).

→ Array data access patterns do not follow row-oriented or columnar patterns. Likely requires new execution engine.

VECTOR DATABASES

Document DBMSs with specialized indexes for (approximate) similarity search on 1D arrays.

→ Vectors represent embedding of corresponding object.

```
{vector: [values],
 metadata: {key: value}*}
```

Pinecone
Weaviate
milvus
qdrant

Vector DBMSs:

→ Accelerate approximate nearest neighbor search via indexes.

→ Not meant to be primary / database-of-record storage.

VECTOR DATABASES

The vector model is not a substantial deviation from existing models that requires new DBMS architectures.

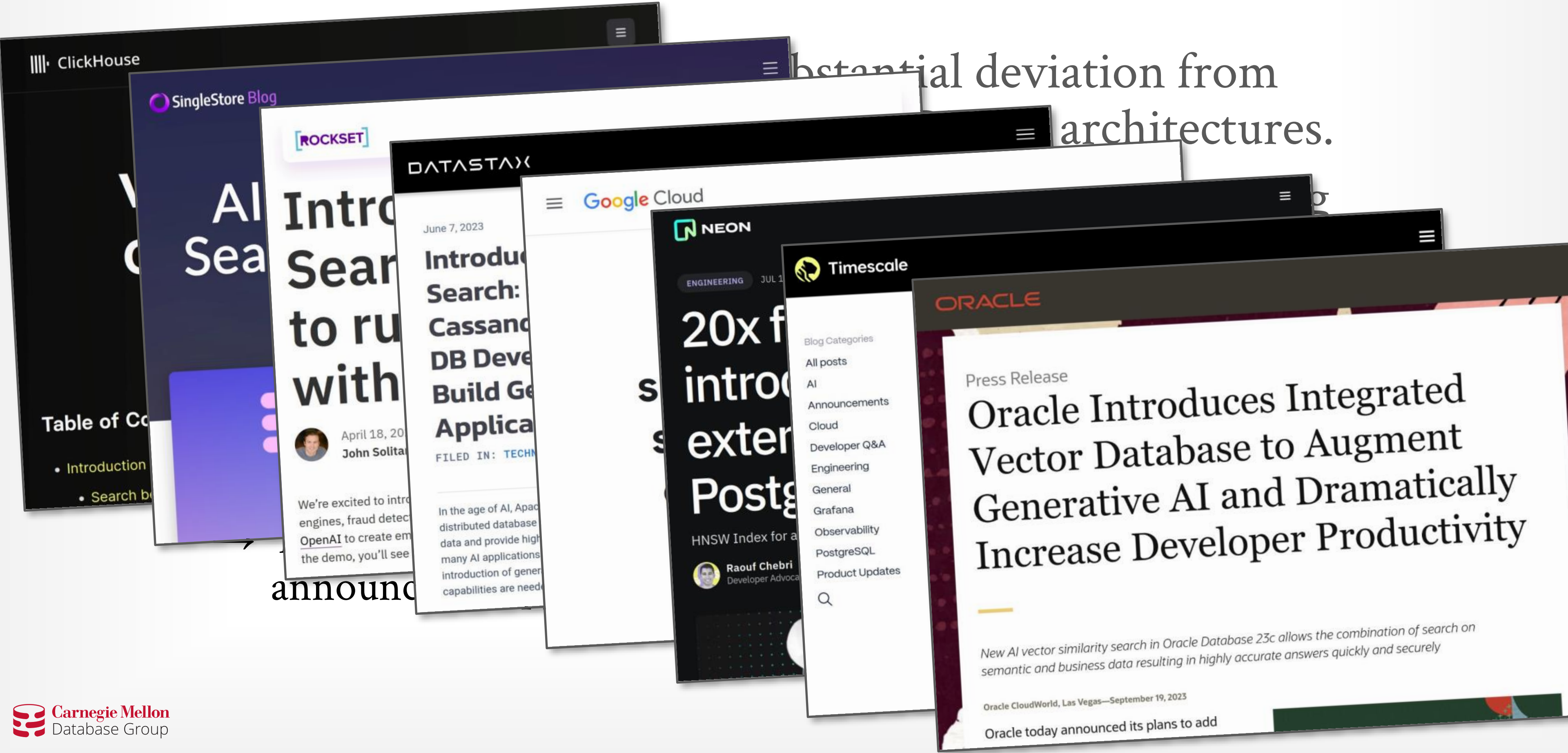
Vector DBMSs offer better integration with AI tooling ecosystem (e.g., OpenAI, LangChain).

Discussion:

→ Every major DBMS will provide native vector index support in the near future.
→ The time from "ChatGPT Buzz" (Q4'22) to existing DBMSs announcing support for vectors (Q3'23) is telling.

VECTOR DATABASES

...bstantial deviation from
...architectures.



announc...

RELATIONAL IS NOT PERFECT

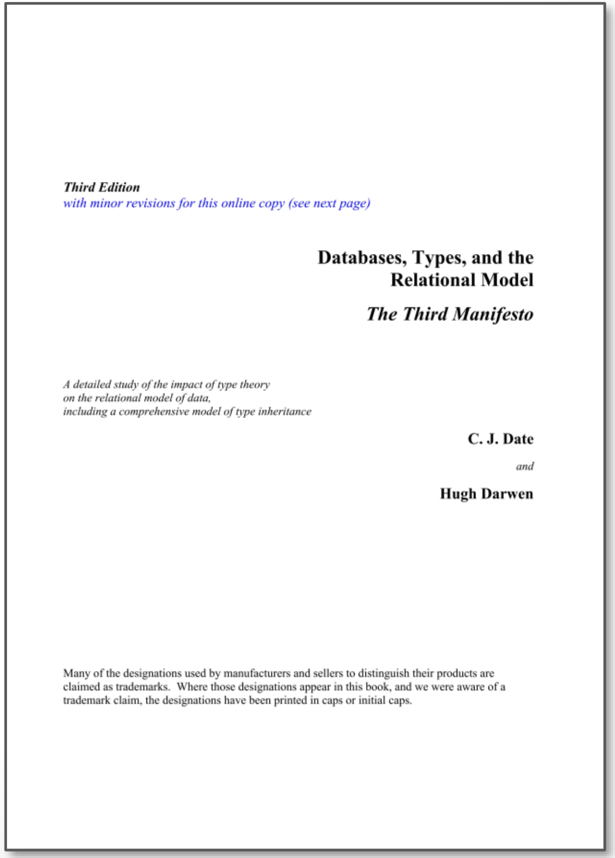
Many non-relational DBMSs provide a better "out-of-the-box" experience than relational DBMSs.
→ Pandas / Jupyter notebooks are still more popular.

Relational DBMS developers should strive to make their systems easier to use and adaptive.
→ Cloud DBaaS hide much of the provisioning / configuration for high availably and durability.
→ DuckDB is very good at this.

SQL IS NOT PERFECT

The deficiencies and problems with SQL are well documented and understood since the 1980s.

The problem of replacing SQL is not a technical problem, but rather it is with overcoming the inertia and proliferation.
→ There is no IBM "juggernaut" anymore…



Third Edition
with minor revisions for this online copy (see next page)

Databases, Types, and the Relational Model

The Third Manifesto

A detailed study of the impact of type theory on the relational model of data, including a comprehensive model of type inheritance

C. J. Date

and

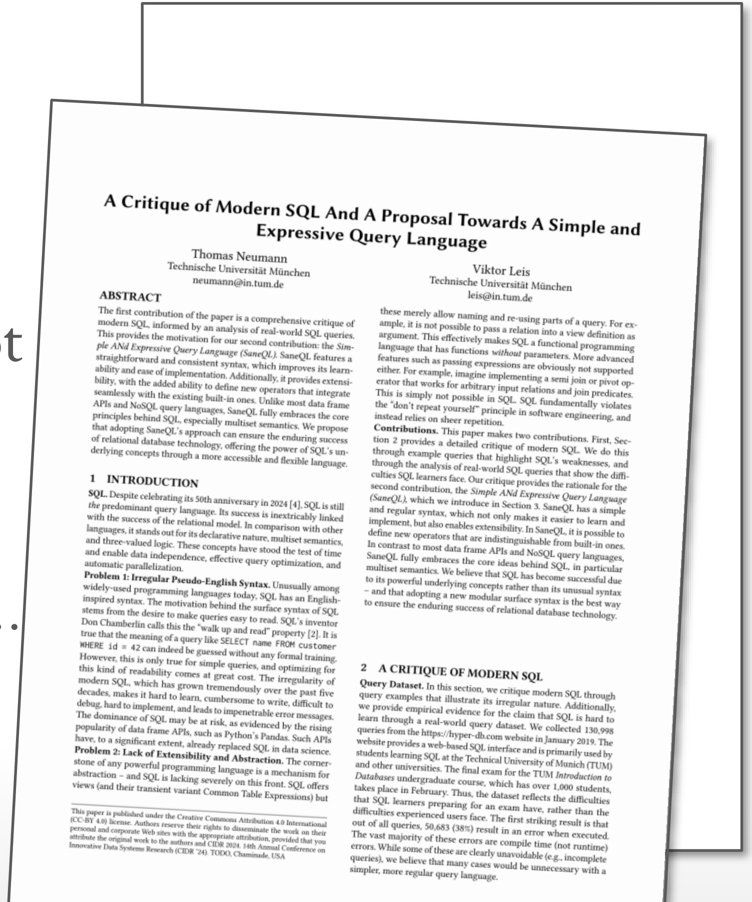
Hugh Darwen

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and we were aware of a trademark claim, the designations have been printed in caps or initial caps.

SQL IS NOT PERFECT

The deficiencies and problems with SQL are well documented and understood since the 1980s.

The problem of replacing SQL is not technical problem, but rather it is with overcoming the inertia and proliferation.
→ There is no IBM "juggernaut" anymore…



A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

The first contribution of the paper is a comprehensive critique of modern SQL, informed by an analysis of real-world SQL queries. This provides the motivation for our second contribution: the *Simple ANd Expressive Query Language (SaneQL)*. SaneQL features a straightforward and consistent syntax, which improves its learnability and ease of implementation. Additionally, it provides extensibility, with the added ability to define new operators that integrate seamlessly with the existing built-in ones. Unlike most data frame APIs and NoSQL query languages, SaneQL fully embraces the core principles behind SQL, especially multiset semantics. We propose that adopting SaneQL's approach can ensure the enduring success of relational database technology, offering the power of SQL's underlying concepts through a more accessible and flexible language.

1 INTRODUCTION

SQL. Despite celebrating its 50th anniversary in 2024 [4], SQL is still *the* predominant query language. Its success is inextricably linked with the success of the relational model. In comparison with other languages, it stands out for its declarative nature, multiset semantics, and three-valued logic. These concepts have stood the test of time and enable data independence, effective query optimization, and automatic parallelization.

Problem 1: Irregular Pseudo-English Syntax. Unusually among widely-used programming languages today, SQL has an English-inspired syntax. The motivation behind the surface syntax of SQL stems from the desire to make queries easy to read. SQL's inventor Don Chamberlin calls this the "walk up and read" property [2]. It is true that the meaning of a query like `SELECT name FROM customer WHERE id = 42` can indeed be guessed without any formal training. However, this is only true for simple queries, and optimizing for this kind of readability comes at great cost. The irregularity of modern SQL, which has grown tremendously over the past five decades, makes it hard to learn, cumbersome to write, difficult to debug, hard to implement, and leads to impenetrable error messages. The dominance of SQL may be at risk, as evidenced by the rising popularity of data frame APIs, such as Python's Pandas. Such APIs have, to a significant extent, already replaced SQL in data science.

Problem 2: Lack of Extensibility and Abstraction. The cornerstone of any powerful programming language is a mechanism for abstraction – and SQL is lacking severely on this front. SQL offers views (and their transient variant Common Table Expressions) but these merely allow naming and re-using parts of a query. For example, it is not possible to pass a relation into a view definition as argument. This effectively makes SQL a functional programming language that has functions *without* parameters. More advanced features such as passing expressions are obviously not supported either. For example, imagine implementing a semi join or pivot operator that works for arbitrary input relations and join predicates. This is simply not possible in SQL. SQL fundamentally violates the "don't repeat yourself" principle in software engineering, and instead relies on sheer repetition.

Contributions. This paper makes two contributions. First, Section 2 provides a detailed critique of modern SQL. We do this through example queries that highlight SQL's weaknesses, and through the analysis of real-world SQL queries that show the difficulties SQL learners face. Our critique provides the rationale for the second contribution, the *Simple ANd Expressive Query Language (SaneQL)*, which we introduce in Section 3. SaneQL has a simple and regular syntax, which not only makes it easier to learn and implement, but also enables extensibility. In SaneQL, it is possible to define new operators that are indistinguishable from built-in ones. In contrast to most data frame APIs and NoSQL query languages, SaneQL fully embraces the core ideas behind SQL, in particular multiset semantics. We believe that SQL has become successful due to its powerful underlying concepts rather than its unusual syntax – and that adopting a new modular surface syntax is the best way to ensure the enduring success of relational database technology.

2 A CRITIQUE OF MODERN SQL

Query Dataset. In this section, we critique modern SQL through query examples that illustrate its irregular nature. Additionally, we provide empirical evidence for the claim that SQL is hard to learn through a real-world query dataset. We collected 130,998 queries from the https://hyper-db.com website in January 2019. The website provides a web-based SQL interface and is primarily used by students learning SQL at the Technical University of Munich (TUM) and other universities. The final exam for the TUM *Introduction to Databases* undergraduate course, which has over 1,000 students, takes place in February. Thus, the dataset reflects the difficulties that SQL learners preparing for an exam have, rather than the difficulties experienced users face. The first striking result is that out of all queries, 50,683 (38%) result in an error when executed. The vast majority of these errors are compile time (not runtime) errors. While some of these are clearly unavoidable (e.g., incomplete queries), we believe that many cases would be unnecessary with a simpler, more regular query language.

SQL IS NOT PERFECT

The deficiencies and problems with SQL are well documented and understood since the 1980s.

The problem of replacing SQL is
technical problem, but rather it is
with overcoming the inertia and
proliferation.

→ There is no IBM "juggernaut" anym

A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL

Jeff Shute, Google, Inc. · Shannon Bales, Google, Inc. · Matthew Brown, Google, Inc. · Jean-Daniel Browne, Google, Inc. · Brandon Dolphin, Google, Inc. · Romit Kudtarkar, Google, Inc. · Andrey Litvinov, Google, Inc. · Jingchi Ma, Google, Inc. · John Morcos, Google, Inc. · Michael Shen, Google, Inc. · David Wilhite, Google, Inc. · Xi Wu, Google, Inc. · Lulan Yu, Google, Inc.

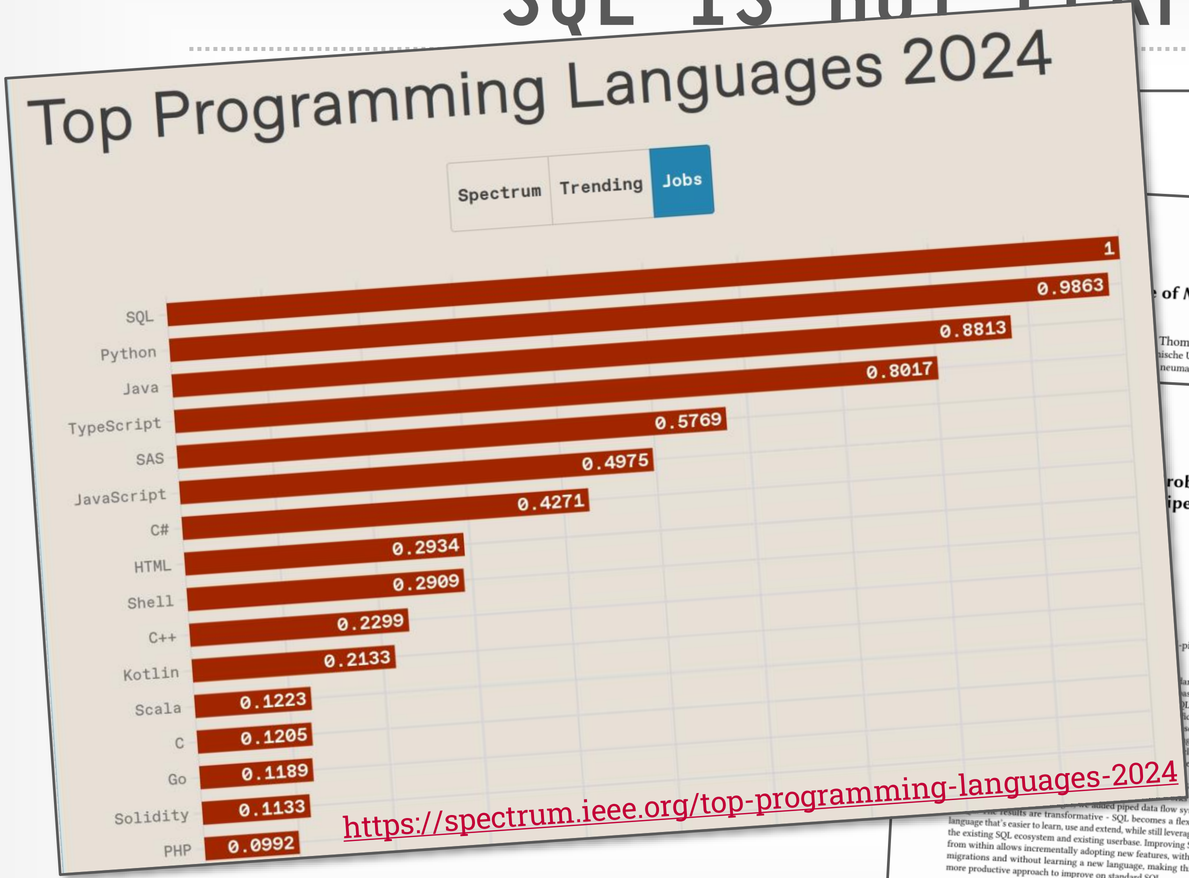
sql-pipes-paper@google.com

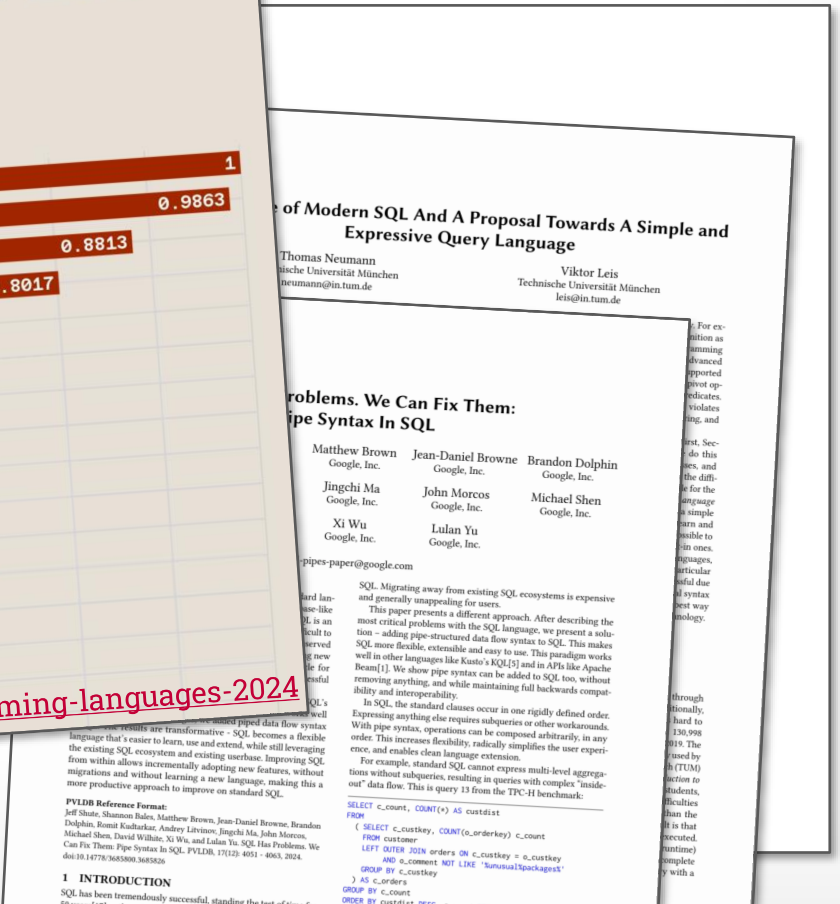
ABSTRACT

SQL has been extremely successful as the de facto standard language for working with data. Virtually all mainstream database-like systems use SQL as their primary query language. But SQL is an old language with significant design problems, making it difficult to learn, difficult to use, and difficult to extend. Many have observed these challenges with SQL, and proposed solutions involving new languages. New language adoption is a significant obstacle for users, and none of the potential replacements have been successful enough to displace SQL.

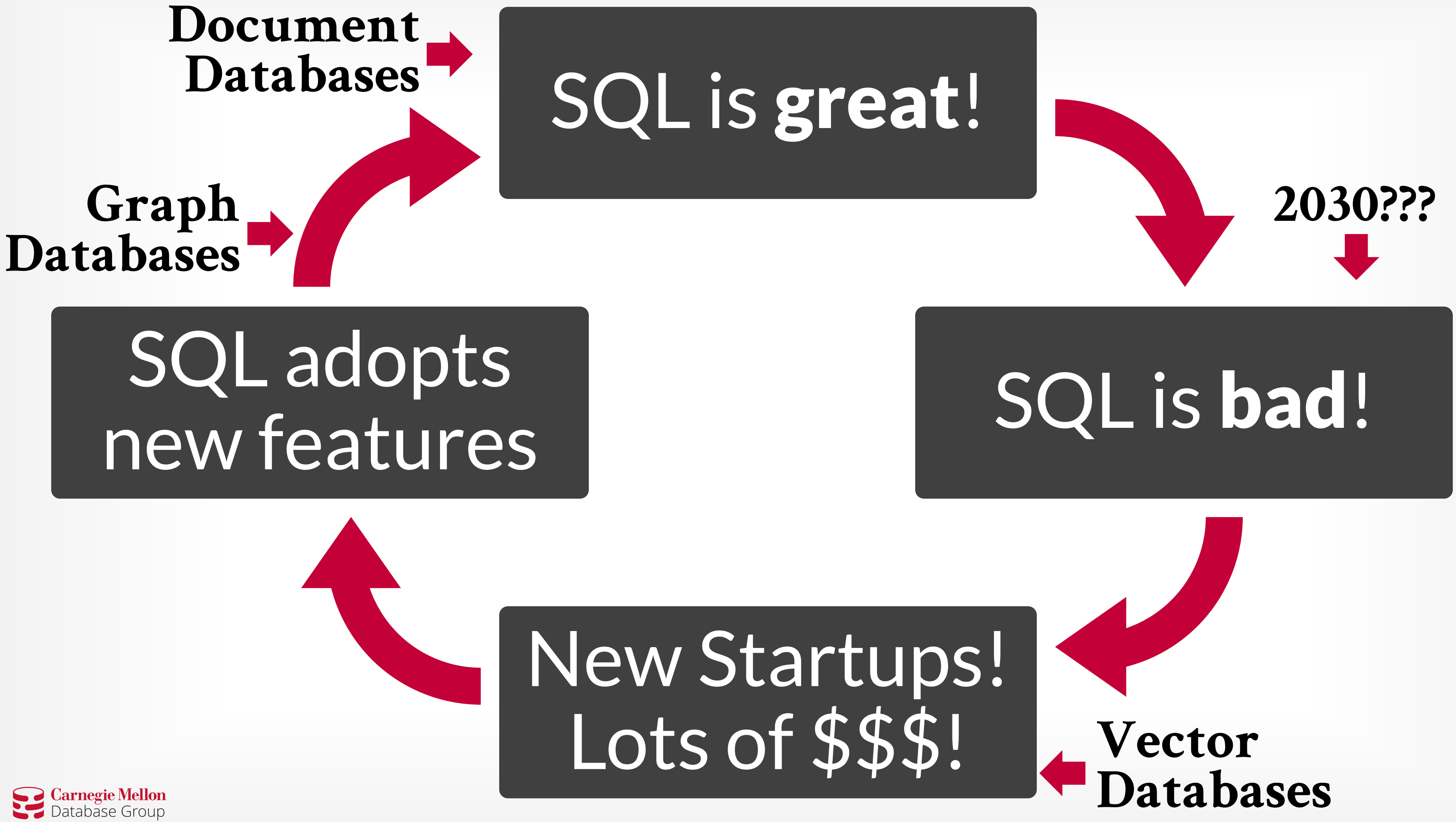
In GoogleSQL, we've taken a different approach - solving SQL's problems by extending SQL. Inspired by a pattern that works well in other modern data languages, we added piped data flow syntax to SQL. The results are transformative - SQL becomes a flexible language that's easier to learn, use and extend, while still leveraging the existing SQL ecosystem and existing userbase. Improving SQL from within allows incrementally adopting new features, without migrations and without learning a new language, making this a more productive approach to improve on standard SQL.

SQL IS NOT PERFECT





https://spectrum.ieee.org/top-programming-languages-2024


SQL is **great**!
SQL is **bad**!
New Startups!
Lots of $$$!
SQL adopts new features
Document Databases
Graph Databases
Vector Databases
2030???

Document Databases

snowflake

DuckDB

Vector Databases

PARTING THOUGHTS

People will continue to make the same mistakes in future DBMS projects.

The demarcation lines of DBMS categories will continue to blur over time as specialized systems expand the scope of their domains.

The relational model and declarative query languages promote better data engineering.

END

Email: pavlo@cs.cmu.edu
Bluesky: @andypavlo.bsky.social
Twitter: @andy_pavlo
Mastodon: @andy_pavlo@discuss.systems

Mike Stonebraker
81st Birthday
October 11th

WHAT ABOUT SQL TRANSPILERS?

Developer-centric frameworks that convert DSL to SQL.

→ Use existing DBMS (PostgreSQL) instead of creating a system just for the language.

No different than ORMs.

Useful for rapid prototyping and ad-hoc projects.

EDGE|DB

MALLOY

PRQL

0xide

convex