



DO WE STILL NEED PEOPLE TO WRITE DATABASE SYSTEMS?

MARCH 2022



Specialized DBMSs for analytics around since the 1970s.

The OLAP DBMS landscape flourished in the 2000s because more organizations had large 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In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common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 claim. 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 "just 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



Columnar Data Storage ➡ **5 Years**
– *C-Store (VLDB 2005)*

Vectorized Query Execution ➡ **10 Years**
– *MonetDB/X100 (CIDR 2005)*

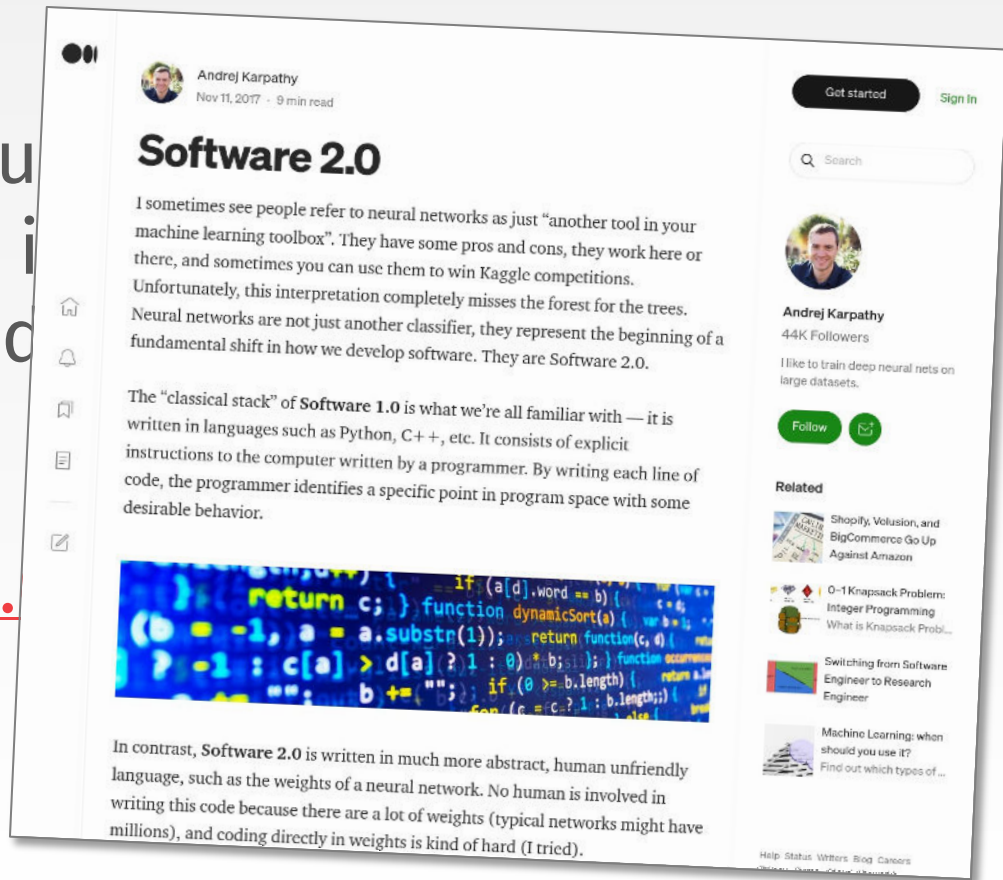
Query Codegen / JIT Compilation ➡ **8 Years**
– *HIQUE (ICDE 2010)*

Can we instead use
Writing a DBMS is hard...
machine learning?



A learned component uses previous observations to predict future behavior instead of a pre-devised strategy.

Example of Software 2.0.





Execution

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



Query Planning

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



Data Storage

- *Compression*
- *Sampling*
- *Caching*



SEDUCTIVE MACHINE LEARNING LEARNED INDEXES

7

Traditional Index

GET(val>567)
Assumptions:

```
SELECT COUNT(*)  
FROM X WHERE val > 567;
```

Read-Only
Known Min/Max



→ **Sorted Data**



Hist-Tree: Those Who Ignore It Are Doomed to Learn

Andrew Crotty
Brown University
crottyan@cs.brown.edu

MOD'18, June 16-15, 2018, Houston, TX, USA

Structures

Frey Dean Google, Inc. Neoklis Polyzotis Google, Inc.

ABSTRACT

Learned indexes in self-organized provide recent results have dramatic data structure improvements to achieve better than the best of the best. In this paper, we present some basic anatomy of learned indexes, and range of the data point space based on learned index in terms of lookup time. We compare learned index to B-trees, in se-

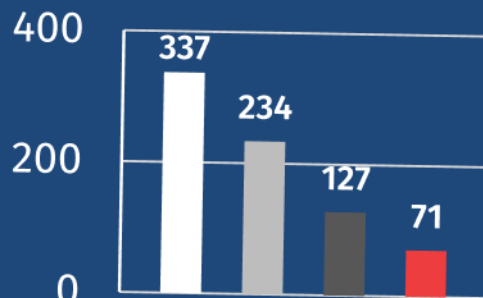
1. INTRODUCTION

Learned indexes can be viewed as the self-organized point space and applied to the data points [12, 13], as well as the components [12]. To demonstrate the usefulness of learned indexes, we compare learned index to B-trees, in se-

This article is published by Morgan Kaufmann and reproduction is provided that you submit a 250-word abstract to the January 10, 2018, CSE

- Binary Search
- B+Tree
- Learned Index
- Hist-Tree

Lookup Time (ns)



OpenStreetMaps
200M Coordinates



SEDUCTIVE MACHINE LEARNING OPTIMIZER COST MODEL

Query Optimizer

Nested Loop Join

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

Alternative Query Plans

LEO – DB2's LEarning Optimizer

Michael Stiller^{2*}, Guy Lohman¹, Volker Markl¹, Mokhtar Kandil²
¹IBM Almaden Research Center 650 Harry Road, K55B1 San Jose, CA, 95139 USA
²IBM Canada Ltd. 1150 Eglinton Ave. E. Toronto, ON M3C 1H7 Canada
 mstiller@csibel.com, {lohman, markl}@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 various assumptions that may or may not be true for a given database. In this paper we introduce LEO, DB2's new optimizer, as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed execution plans, LEO compares the optimizer's estimates with 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 on-line or off-line on a separate can be done either incrementally or in batches. In this system, and other incremental or in batches. In this system, 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, as 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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the TDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.
 Proceedings of the 27th VLDB Conference,
 Roma, Italy, 2001

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 30 years [SAC+79, Ge93, S894, ARM89, Ly88]. 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 statistics:** 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 subsets of columns to maintain multi-dimensional histograms.

* 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

Nesime Tatbul
MIT & Intel Labs
tatbul@csail.mit.edu

Parimarjan Negi
MIT
pnegi@csail.mit.edu

Mohammad Alizadeh
MIT
alizadeh@csail.mit.edu

Hongzi Mao
MIT
hongzi@csail.mit.edu

Yingwei Li
MIT
yingwei@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization 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].

KEYWORDS
 query optimization; machine learning; reinforcement learning

1. Reference Format:
 Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Yingwei Li. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, June 20-25, 2021, Virtual Event, China. ACM, New York, USA, 14 pages. <https://doi.org/10.1145/3446016.3446018>

INTRODUCTION

Optimization is an important task for database management systems. Despite decades of study [79], the most important elements of query optimization – cardinality estimation and cost modeling – remain difficult to teach [4]. Several works have applied machine learning techniques to these stubborn problems [37, 46, 44, 39, 72, 73, 74, 76]. While all of these new solutions demonstrate promising results, we argue that none of the techniques are yet practical, 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 [49, 51, 53, 39].
- (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, 46]. 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 (Bao: Making Learned Query Optimization Practical) 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://github.com/ryanmarcus/bao>



This work is licensed under a Creative Commons Attribution-International 4.0 International License.
 See <https://creativecommons.org/licenses/by/4.0/> for details.
 Copyright © 2021, Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Yingwei Li.
 DOI: 10.1145/3446016.3446018



Failsafe Mechanisms

- *What do you do when models are horribly wrong?*

Explainability

- *How to tell humans why DBMS made certain choices?*

Human Feedback / Overrides

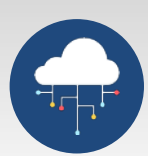
- *Can a human provide hints? What if they're wrong?*

Transferability

- *Can we reuse knowledge gained from one database and apply it to another?*

*Can we instead use
machine learning?*

Yes, but...



There are already ML-powered tools to optimize database instances.

- *Leverage existing APIs to extract telemetry and apply changes to DBMS.*

Classic Database Administration

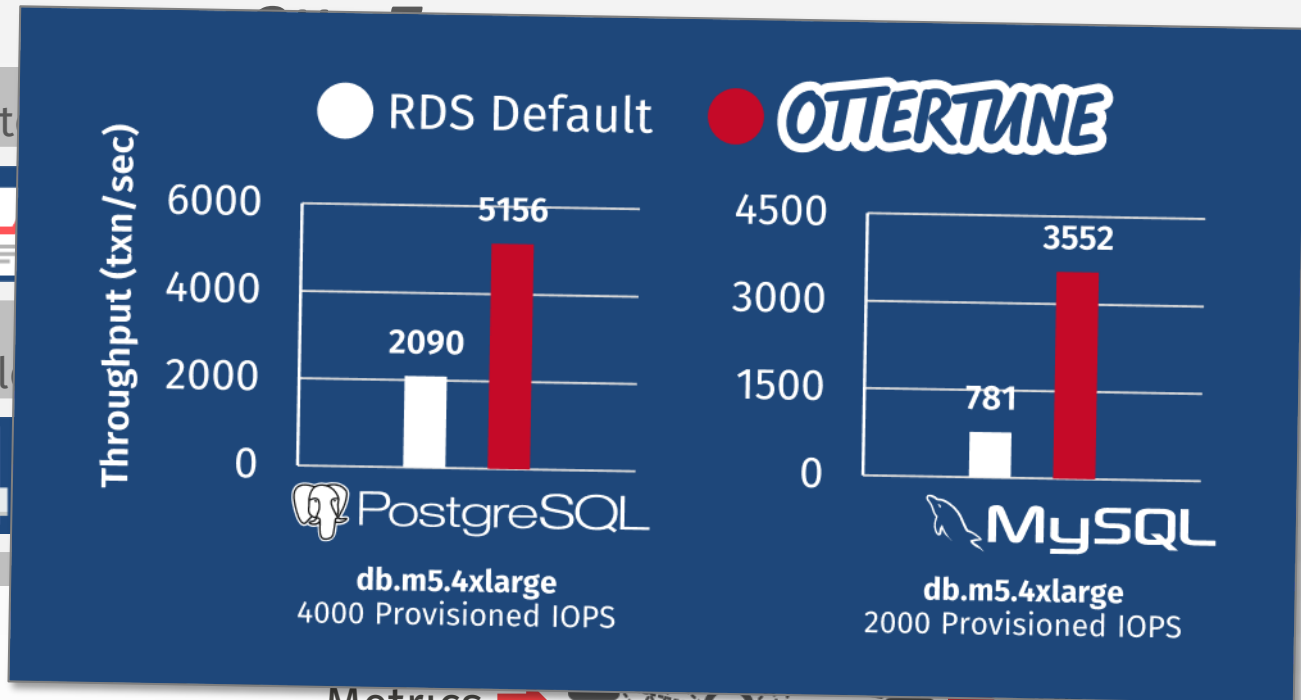
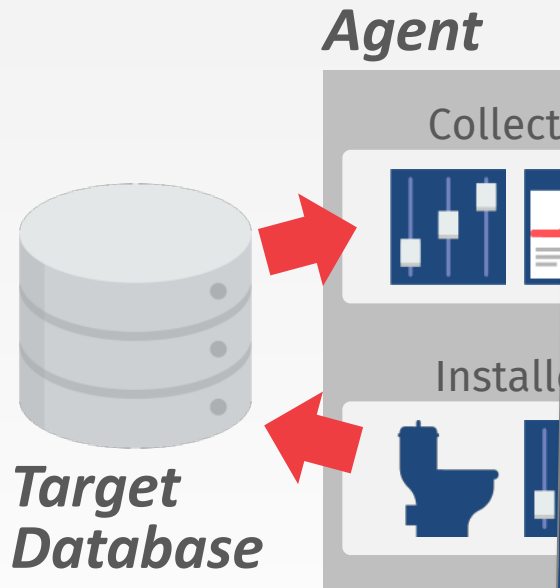
- *Physical Database Design (Indexes, Partitioning)*
- *Knob Configuration*
- *SQL Tuning*



ML EXTERNAL TOOLS

AUTOMATIC CONFIGURATION TUNING

12



***What Does the Next
20 Years Look Like?***



Challenge #1:

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

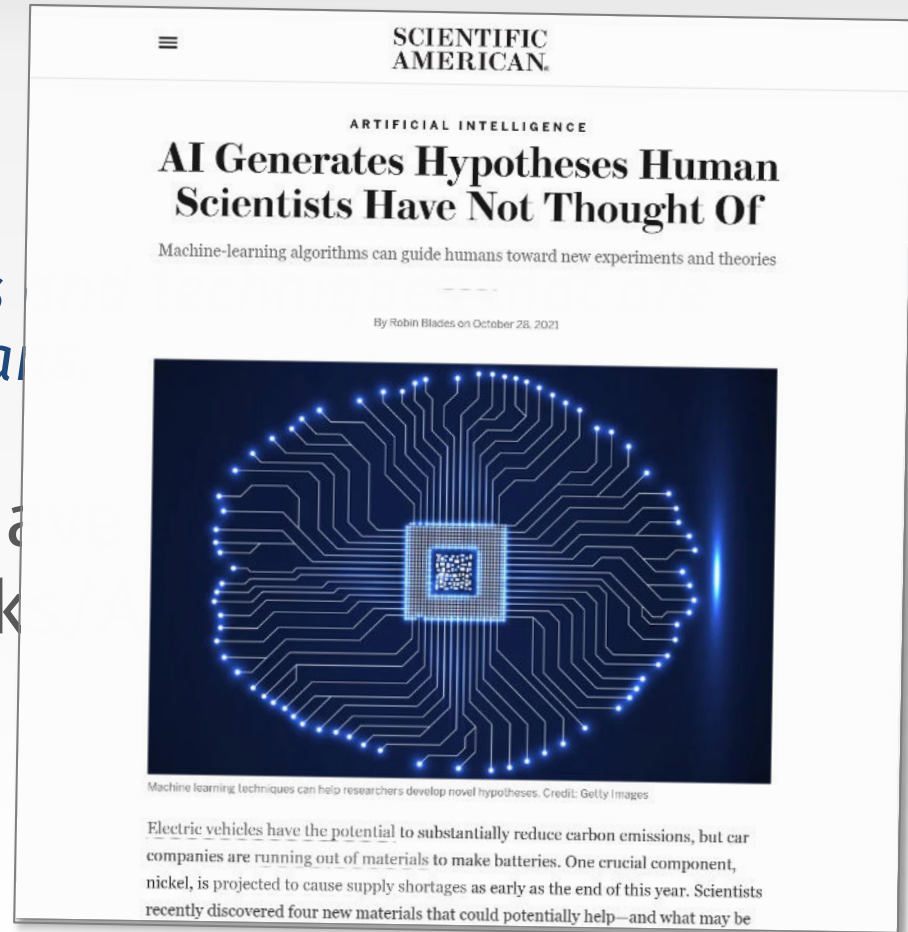
Existing automation methods are reactive.
Humans are also proactive.



Challenge #2:

- *Discover new optimizations currently unknown to humans*

This requires a DBMS to have
and instrumentation hooks





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

- *Still require human experts to understand how leverage ML properly in the system.*

The next challenge is how to use ML to find things beyond human thoughts.

The background of the slide is a dark blue color with a sunburst or radial pattern emanating from the center. The pattern consists of many thin, light blue lines radiating outwards, creating a starburst effect.

END

@andy_pavlo