ML-In-Databases: Assessment and Prognosis

Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M. Patel, Chris Ré, Michael Stonebraker

Abstract

With the rapid adoption of Machine Learning (ML) in Computing, there has been a flurry of recent research considering using ML to build the internal component of database systems. While initial work in this area has shown interesting results, the jury is still out on whether these methods will replace existing methods. A group of researchers with opinions on both sides of this issue met to assess the state of this area and to formulate a plan for the next steps that would be needed to determine the potential role of these new ML-based methods in building future database systems. This article summarizes the collective perspectives that resulted from these discussions. First, this article describes broad forces that are changing the landscape in which database systems are deployed, connecting several trends that likely require rethinking how future database engines are built. Next, this article describes the different perspectives on this topic of using ML methods to replace existing internal database components. Finally, the key takeaways from this discussion are presented, and these takeaways also point to directions for future research.

1 Introduction

Enterprises increasingly want to move to making data-driven decisions across every aspect of their business. To achieve this goal, they are increasingly adopting a rich ecosystem of data management tools, and database systems are invariably at the heart of these data ecosystems. However, we see four, possibly very disruptive forces, that are likely to strongly influence future data systems. These forces, which range from well-established ones (force #4) to some in their infancy (forces #1 and #2), are described next.

1.1 Force #1: The changing end-user landscape

First, from the end-user perspective, there is a clear goal in enterprises to democratize data, moving to holistic data-driven decision making across the entire enterprise. Essentially, enterprises want *every* business user in their organization to make data-driven decisions, and often in real time.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

This transformation will require that a broad range of employees in an organization must have access to the underlying enterprise data (with appropriate access controls and often via applications). In this new world, the end user is a non-programmer business user, as opposed to a business analyst, who asks ad hoc questions using a variety of querying paradigms, including natural language and no-code interfaces.

With this driving force, the "audience" for a database system will explode from database programmers and business analysts to a much larger group of business users. This driving force will also change the technical programming skill set that one can expect from an end user.

1.2 Force #2: The changing application surface

Second, in this new data-democratized world that we see coming, end users will demand that the underlying platform carry out a broader range of data analysis tasks. These tasks go beyond traditional structured query processing (SQL) to include machine learning (both exploratory and predictive) and more complex analytics, such as regressions and principal components analysis. Moreover, we expect sophisticated visualizations and natural language processing (NLP) to become much more important. Further, these tasks must be carried out efficiently on an ever-increasing volume of data.

In addition, when deploying machine learning (ML) for prediction, there is a critical need (increasingly driven by regulations) to manage and archive the entire data pipeline that is associated with a deployed ML model. In other words, there is a need for data platforms to provide robust model management as an integrated core feature of the data platform itself.

1.3 Force #3: The changing data organization

Third, an often-quoted anecdote is that data scientists spend at least 80% of their time finding data sets they want to analyze, cleaning the data and integrating it together. A further anecdote is that independently constructed data sets never have a common schema, which makes the data integration problem noted above especially difficult. Obviously, analyses whether through ML or other means will produce garbage on dirty or inconsistent data. As a result, a key ground stake for this new data-democratized reality is that there must be a single source of truth for data in the entire enterprise. This aspect is leading to the consolidation of data spread across data silos into a common data repository, often called a data lake. To fully leverage this data lake, we see enterprises making substantially greater investments in data cataloging, data integration and data cleaning off into the future.

1.4 Force #4: The changing hardware substrate

The last driving force, and in many ways the force that is accelerating the other three forces described above, is the move to the cloud. This force fundamentally shifts the substrate on which analytic database software runs. Gone are the days in which a business intelligence database system ran on hardware consisting of a physical server box resident in the enterprise, and consisting of predefined fixed amounts of compute, memory, storage and networking components. In the cloud the server becomes a collection of disaggregated compute, memory, storage (with many more tiers), and networking resources, all of which can be mixed in just about any proportion.

Further, this virtual server can be elastically sized, often in a matter of seconds, to add or subtract any of these hardware resources. While such a virtual server may be a collection of many physical components spread across a physical data center (or data centers), logically, it presents itself as a single server. We call this virtual server a *cloud server* in the remainder of this article.

What is even more powerful is that cloud servers are more economical than traditional servers, as they are priced using a pay-as-you-go paradigm. Enterprises can size up their cloud server capacity for a short amount of time and size it down when done, making internal capital allocation processes far cheaper. Thus, large data projects now become more financially viable.

Traditional database systems are notorious for having a large number of tuning knobs that have to be continually tweaked for performance by human database administrators (DBAs). These knobs have to be set at the "right" values when the database is initially installed, and then re-tuned as the workload changes. This reliance on humans to keep database installations running efficiently often results in sub-optimal performance, costing enterprises time and money. Modern cloud data platforms take this problem head-on and are moving toward self-tuning. Thus, in the cloud era, there will be a huge reduction in the human IT cost associated with running a cloud data warehouse.

We see three consequences of these forces, to which we now turn.

2 Consequences of these forces

The database community has started to make fundamental changes to react and leverage the four driving forces described above. A key direction in recent years has been to rethink the methods that go into every internal component inside a database engine. These components include storage management (which includes indexing), query processing methods, query optimization, and query scheduling. In a traditional data processing system these components are tightly coupled at the architecture level and optimized to run on a specific hardware configuration.

In a cloud server, the storage and compute is disaggregated in the underlying cloud fabric. Each of these components can be increased elastically independent of each other. Fast networking is often implicitly provided by the cloud provider. Further, each component of the storage hierarchy can be provisioned (and adjusted elastically). Thus, the bulk of the data sits in a cloud file system, which in principle has infinite storage capacity. That data can be accessed by compute resources by pulling data from the cloud file system (CFS). Along the way the data can be cached in multiple storage locations, including local SSDs and main memory. By-and-large traditional memory hierarchy rules apply. Storage that is closer to the processor is accessible at a lower latency, but has a higher unit storage cost, and lower capacity compared to storage that is "further away." What is interesting however, is that in a cloud server, every layer of the storage hierarchy can be elastically adjusted. Essentially a data platform can now provision a virtual cloud server in an arbitrary configuration to best suit the target application needs, and then reconfigured easily (and economically) as the workload changes. Database systems that don't leverage this elastic behavior, will simply not survive the competition in the cloud era.

Traditional database systems will need major reworking to achieve this sort of elasticity. Since a cloud server is assembled from a set of disaggregated underlying hardware resources, to make full use of this hardware organization, the internals of database engines must also be disaggregated to run at the speed of the underlying hardware. There is broad consensus in the database community that database internal components and architecture must be rethought from scratch for this new cloud server world.

Another trend which follows from these forces is the Software 2.0 movement [9]. This new approach to software engineering aims to move from thinking about software as a predefined static organization of code to a dynamic model-based view. In this new approach, machine learning models, often a neural network, are a fundamental software building block. This new building block can be viewed as a dynamic function that takes data as input, runs it through the ML model, and outputs the predictions from the model as the output/result of the function. A critical aspect of this new approach, which we call *ML-functions* (*MLF*s), is that an MLF can easily be retrained using new data. Thus, an MLF can dynamically optimize itself to adapt to new workload characteristics. This adaptation can often be done internally in the MLF providing an elegant software abstraction in which the software system can be considered to be a collection of dynamic, self-optimizing components. MLFs provide a disaggregated approach to software engineering and using it purposefully in this manner provides an interesting parallel to the disaggregation of hardware components in a cloud server. We expect complex software systems (not just database systems) will move toward this paradigm to reap the benefits of self-optimization.

Synergistic with the Software 2.0 approach is the move to serverless computing. With serverless computing, provisioning of hardware resources is automatically done by the serverless infrastructure freeing the developer from having to worry about that aspect of application development. Software 2.0 provides a new software abstraction while serverless provides a new hardware abstraction, and together both make the life of a modern software developer easier.

In summary, we are already seeing DBMSs being rearchitected for cloud deployment. In addition, serverless computing, such as AWS Lambda, are becoming an important cloud platform. Finally, the benefits of self-adaptation are obvious, which will drive large software systems, including DBMSs, toward using ML.

3 ML-in-Databases

Against the backdrop of changes in the broader data landscape described above, a panel was organized at the 2021 Annual Conference on Innovative Data Systems Research (CIDR). The topic for this panel was the role of Machine Learning (ML) in driving transformative changes to database internals. Below we summarize the discussion that started at the panel and continued after the initial panel discussion, resulting in the discussion and direction that is described in this article.

3.1 Learned indices

The first topic for discussion was the role of learned methods to reimagine a variety of database internal components. There has been a flurry of work over the last few years advocating the use of ML models to replace the traditional database internal components. This line of work can be considered as advocating for a Software 2.0 approach to reimagining database internal components. There have been two big themes in the community, namely learned indices and learned query optimizers. We discuss learned indices in this section, and we cover learned query optimization in the next section.

The learned index approach challenges the way in which database systems use indices. The initial focus of this line of research (e.g., [10, 7]) has been to provide an alternative to the "ubiquitous" B+-tree index structure [5]. In addition, there has been early work on other index structures, including LSMs and spatial indices. To focus this article, we only discuss the work related to the B+-tree index.

In a traditional database system, search on a large dataset can be sped up by building a B+-tree index. A B+-tree index is a disk-optimized balanced tree data structure that has a $log(N)$ average case search time and I/O complexity. However, one can view a B+-tree search operation as a lookup function that takes as input an argument (the search key) and returns the set of matching records from the underlying dataset as a result. In a B+-Tree index, this function has a large data structure (the index) that is associated with the search function. A learned index on the other hand can be viewed as replacing this data structure with a MLF, and often a hierarchical collection of MLFs.

The panel discussion on this topic focused on past approaches to learned B+-tree index structures and quickly pivoted to the central question: *Do these learned indices actually outperform traditional B+-trees, especially when one uses optimized B+-tree indices?*

Over the last four decades, a number of B+-tree optimizations have been proposed, including cache-efficient implementations, and use of sophisticated key compression methods. It was pointed out that both prefix and postfix compression methods have been used in VSAM indices [3], which have been in commercial systems for over four decades. These optimizations often produce huge efficiency improvements. Thus, comparison with an optimized B+-tree implementation that also includes considerations, such as a broad range of compression techniques, is needed to better understand the role of learned indices in practice.

It was also pointed out that while the research enthusiasm has been high for learned indices, practical adoption will take time and the final judgement about their practicality is still out. The early work on learned indices has largely ignored issues of concurrency control mechanisms, and non-main memory storage. Indices are typically common in transactional workloads and robust concurrency control methods are ground stakes for commercial adoption. There is only preliminary work to-date on how to make learned index structure work with a mix of update and read-only queries, and comparison with non-main memory database systems is missing. Further, the issue of crash recovery has not been addressed by work on learned indices, and that too is crucial for commercial traction.

The proponents of learned indices argued that comparisons with highly optimized B+-trees for in-memory settings including prefix compression (e.g., against the ART index) have been carried out [10, 12] but that other settings (disk, concurrency) are still missing. However, one should view the work on learned indices as work in progress.

The proponents of learned indices also pointed out that a critical missing component is the lack of realistic workloads, which is a broader community challenge. For example, if the data being indexed is Gaussian distributed, then a MLF will outperform a traditional index. Thus, the performance tradeoffs are very dependent on the distribution characteristics of the search keys. Getting realistic data distributions has been challenging, and this aspect is critical in determining the pros and cons of the two approaches to indexing data.

On the issue of performance of learned indices, there is a change in emphasis in parts of the learned index community from targeting execution time performance improvements to targeting space efficiency. Early results indicate that the space complexity of learned indices is provably better than B+-tree indices [8], and some commercial systems (e.g., Google Bigtable) observe throughput improvements because of the smaller index size [1]. Similarly, the smaller (and faster) lookup performance of learned indexes improved the end-to-end throughput of LSMs, like RocksDB [6]. However, the practical benefits depend on the workload, making good workloads/benchmark central to also moving this aspect of the learned index research forward.

3.2 Learned optimizers

Learned query optimizer is another dominant contemporary research theme in the database community. This sub-community proposes an learned alternative to traditional query optimization (e.g., [11, 14]). A traditional query optimizer takes as input a query statement (often an algebraic expression) and transforms it to an optimal plan (another annotated algebraic expression). The internal code organization of a traditional query optimizer is often a large mesh of functions (with some functions being rules-based) that work together to formulate an optimal execution plan.

Query optimization is a notoriously difficult task and one that after five decades of work continues to drive significant research attention. A learned approach to query optimization takes a Software 2.0 approach to this internal component of a database engine, and aims to use an MLF, often backed by a deep neural network, to optimize queries.

Similar to the discussion above on learned indices, it was pointed out that work on learned query optimizers is also preliminary, and there isn't conclusive evidence that such approaches are yet practical for real deployments. Besides, query optimization is really critical in analytic environments, and state-of-the-art systems use a column store organization in such settings, as column stores result in much better query performance. However, until recently all comparisons of learned query optimizers were against the PostgreSQL optimizer or traditional row-oriented optimizers (e.g., Oracle or Microsoft SQL Server). Showing gains for analytic queries on a row-store database does not make a strong case for learned query optimizers.

Proponents of the learned methods pointed out that existing work already compares against some of the best-known optimizers, such as Microsoft SQL Server and Oracle [14, 13], and noted that PostgreSQL is widely used as it is open-sourced. This open-source aspect is important as the research in this area requires making changes to the actual optimizer source code to implement the learned optimization methods. It was further noted that commercial systems have started to integrate learned cardinality estimates and other ML-based improvements.

Proponents of the learned methods also highlighted that there is an additional benefit of pursuing research in learned methods, which is to explore it as a research direction to determine what its boundaries are. Even if at the end of the day, they do not end up outperforming state-of-the-art systems, we will as a community learn a lot from this process. Thus, there is a pure research exploration component to pursuing learned approaches.

3.3 Instance optimality

The diversity of database workloads and the elastic nature of the cloud server on which a database instance runs, presents unique opportunities to tune a specific database instance to the workload and hardware parameters available at that instant. This idea of instance optimal database deployments is an emerging research trend in the database community. ML methods are clearly applicable here as many database systems have hundreds if not thousands of parameters, and the choice of the parameters plays a big role in determining the overall system performance. Further, even if the parameters have been set once, as the workload changes these parameters must be reset.

Today such database tuning is largely done statically. However, there is a growing body of research work in our community that advocates taking as input data from the operational environment – think database and system logs – and using it for self-tuning. There has been sporadic work over the last four decades on self-tuning, and ML-based algorithms are likely to play a role in this endeavor, if they can be shown to be superior to traditional methods.

3.4 ML-in-Databases, the "other" ML

This panel of researchers also noted that perhaps the real reason to bring ML methods into database engines is orthogonal to replacing existing internal database data structures/algorithms with learned parts. This reason is related to the rapidly changing database workloads and user base (recall Forces #1 and #2).

Database workloads are changing quite rapidly. With a unified data-centric view of the enterprise (Force #3) enterprises can run both analytic and ML workloads in the same data engine. Today, these two types of workloads are often run in two different platforms. For example, a platform that runs SQL analytics and another that runs ML analytics (e.g., Pandas or Tensorflow). Managing two platforms instead of one increases the data management costs and also makes data governance more complicated. This current approach of "ML-outside-the-database" is a huge and growing issue for enterprises.

The community has worked on the integration of ML and SQL in the past. Some commercial database systems (e.g., [4]) have been shipping ML methods that run inside the database engine since the earlier part of this century. However, what is needed is a dramatic new approach to not just make ML and SQL workloads operate in the same data engine, but also to support ML model management in the data platform. Today, there is an unmet need for robust management of ML pipelines (simplistically think of this as an "execution plan" with data cleaning, encoding, training, and scoring "operators"). To use an ML model for inferencing (e.g., making a credit recommendation for a new loan applicant), the enterprise has to put in place a vast data management infrastructure for the ML models. Regulatory or internal governance policies may require that the entire pipeline associated with creating the deployed ML model be archived so that it can be revisited in the future (e.g., to check for bias). As the underlying data changes, a previously optimal model may "drift," and may also need to be retrained to make more accurate inferences, which again requires support for robust model management.

In addition, if one looks at the life cycle of data, especially when an enterprise is trying to bring all data together in an enterprise-wide data lake (Force #2), one sees that there is ML and SQL intertwined at every step. Just getting the data into the data lake requires data cleaning, which will require using ML methods at scale. Further, this data cleaning "workload" is not a one-time job, but it is a workload that must be run continually. Users (and now there is a broader user base due to Force #1) also invoke analytics functions that often use ML for data exploration and visual discovery. In fact, in many enterprise data settings, these new data workloads (data cleaning and discovery) take up most of the time of the human user as well as most of the resources in the cloud server. When taking a holistic look at the workload present in a modern data ecosystem, the ML part is more dominant than traditional SQL.

Thus, the real opportunity for ML and database systems is to bring both ML processing and SQL processing together into the database engine. To achieve this goal, we need to rethink the internal organization of a database engine even more, and not just try to use ML/learned methods to speed up a traditional database platform, which arguably only supports SQL and thus a small fraction of the overall enterprise data workload.

Overall, a strong sentiment expressed in the panel was that the real synergies for ML and database to come together may lie in this area – namely, creating new data technologies that efficiently serve both workloads natively, and also provides in-built model management capabilities.

4 Key Takeaways

The discussion on this topic of ML-in-databases can be summarized by the following key takeaways, which also points to directions for future work.

Takeaway #1: The initial comparisons of leaned indices with optimized traditional indices should be further expanded to include concurrency control and multi-user settings. Learned indices need to make a holistic argument if they want to challenge the use of traditional indices.

Takeaway #2: A key benefit of learned indices may come from the learned structures requiring lower space utilization, rather than a reduction in search time. Further work in this area is needed to test this claim.

Takeaway #3: More realistic benchmarks/workloads are critical as the benefits of learned indices is very dependent on the data distribution. Coupled with Takeaway #1, it may be time for leaned indices to use end-to-end benchmarks in their comparison, with full parity on features like concurrency control, recovery, non main memory, and multi-user settings.

Takeaway #4: When evaluating the benefits of a learned optimizer, the yardstick for comparison should not just be PostgreSQL – a row-store system that is known to have low performance. Since access to the database system source code is required for this research, using an open-source column-store database system, such as MonetDB [2], is advocated.

Takeaway #5: To fully exploit the disaggregated and elastic nature of a cloud server, database deployments will need to consider instance optimal methods so that they can self-tune (and retune) themselves for the workload at hand. ML methods are likely critical here as long as they can demonstrate superiority to traditional methods.

Takeaway #6: The big opportunity for ML-in-Databases is for database systems to consider new data processing techniques/architectures that can efficiently process both ML and SQL tasks. The internal data structures used in such a platform may still be "traditional," but the huge opportunity ahead for the community is to expand the scope of workloads in this manner, and to include model management as a core data management task.

5 Acknowledgments

This work is supported in part (for effort by JMP) by NSF under grants OAC-1835446, IIS-1815701, and by CRISP – one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Hussam Abu-Libdeh, Deniz Altinbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. Learned indexes for a google-scale disk-based database. *CoRR*, abs/2012.12501, 2020.

[2] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[3] IBM Knowledge Center. Key compression. `https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.idad400/comp.htm`. Accessed: 2021-02-12.

[4] Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Scalable classification over SQL databases. In *ICDE*, pages 470–479. IEEE Computer Society, 1999.

[5] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[6] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. In *OSDI*, pages 155–171. USENIX Association, 2020.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: an updatable adaptive learned index. In *SIGMOD Conference*, pages 969–984. ACM, 2020.

[8] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. Why are learned indexes so effective? In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 3123–3132. PMLR, 2020.

[9] Andrej Karpathy. Software 2.0. `https://medium.com/@karpathy/software-2-0-a64152b37c35`, 2017. Accessed: 2021-02-12.

[10] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *SIGMOD Conference*, pages 489–504. ACM, 2018.

[11] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[12] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2021.

[13] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.

[14] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.