Future-Proof Data Systems

Jana Giceva
Technical University of Munich
jana.giceva@cit.tum.de

ABSTRACT

Database systems are increasingly expected to support evolution in data workloads, representations, and hardware environments. Existing designs often respond through either specialization, which can deliver high performance but leads to narrow and brittle systems, or extensibility, which broadens functionality but can produce opaque extensions and growing system complexity.

This paper summarizes a line of work that treats future-proofness as an abstraction-design problem: we identify boundaries across the data processing stack where concerns must evolve independently, and design interfaces that preserve enough structure across that boundary for optimization. We illustrate this at three levels of the stack. At the level of data structures, Sortledton unifies analytics, pattern matching, and transactional updates on a single graph representation. At the level of data formats, AnyBlox allows engines to read datasets that provide sandboxed self-decoding logic, reducing the need for bespoke readers. At the level of query processing, LingoDB and its declarative sub-operator layer recast query optimization as a sequence of extensible compiler passes, enabling cross-domain optimization and hardware-specific lowerings.

PVLDB Reference Format:
Jana Giceva. Future-Proof Data Systems. PVLDB, 19(12): 4965 - 4972, 2026.
doi:10.14778/3827998.3838711

1 INTRODUCTION

Designing data systems that can robustly evolve with changing workloads and hardware environments is becoming increasingly difficult. Modern systems are already under strain with the range of features they can offer and support. At the same time, users increasingly prefer open data formats, federated architectures and modular infrastructures that allow data and processing engines to evolve independently. These trends are accompanied by a rapidly changing hardware landscape, ranging from heterogeneous processors and accelerators, to new memory technologies and fast interconnects.

The conventional responses are specialization and extensibility. A traditional monolithic system design can provide extension points allowing for new features to be added through local interfaces or black-box extensions, but it could likely lead to a highly complex and bloated system, which is then difficult to maintain and debug. Alternatively, one could aim for specialization that can deliver excellent performance, but result in a multitude of narrow systems that are difficult to evolve and adapt. Designing future-proof data systems therefore requires more than adding extension points or accumulating specialized implementations [22].

The central thesis of our work is that future-proofness is an abstraction-design problem. A system cannot anticipate the nature of future workloads, how data formats will evolve, or what hardware platforms will look like in a few years from now, but it can define boundaries that let these concerns evolve independently while preserving the information needed for efficient execution. The goal is not to avoid specialization, but to derive it from abstractions that expose the right structure to optimizers, compilers, and runtimes so the system remains performant yet robust.

We demonstrate this methodology at three layers of the data processing stack. We first study the boundary between workloads and physical data structures through Sortledton [10], a universal transactional graph data structure (§3). We then consider the boundary between data formats and processing engines through AnyBlox [12], a framework for self-decoding datasets (§4). Finally, we discuss the boundary between computational semantics and physical execution through LingoDB [33], its open intermediate representations and extensible optimization and compilation framework [23], and the declarative sub-operators [21] (§5). Together, these projects show how carefully chosen abstractions can make systems more adaptable without giving up performance.

2 BACKGROUND AND MOTIVATION

Future-proof data systems must respond to change stemming from two concurrently moving tectonic plates (fig. 1): evolution of the application's workloads and requirements (§2.1) and the hardware environment they run on (§2.2). Database systems already leverage a broad set of mechanisms to address each plate individually – user-defined functions, plug-in architectures, standardized data representations, and hardware-specific kernels among others. The difficulty, which we discuss in §2.3, is that these mechanisms typically expose interfaces that are local rather than semantic: they let a system add functionality without necessarily allowing the rest of the system to reason about, optimize or re-target.

2.1 Changing usage requirements

Traditional database systems were designed around comparatively stable assumptions: structured relational data, well-defined schemas, and a limited set of transactional and analytical workloads. Modern applications increasingly break these assumptions. They combine multiple forms of computation within the same pipeline, such as relational analytics, graph processing, ML inference, user-defined functions, and domain-specific transformations. As a result, the system can no longer assume that all relevant computation is expressible through a fixed set of relational operators.

Data usage changed as well. The same dataset may be accessed by a database engine, a data-lake distributed query engine, or an ML framework. Users therefore expect data to remain portable

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3838711

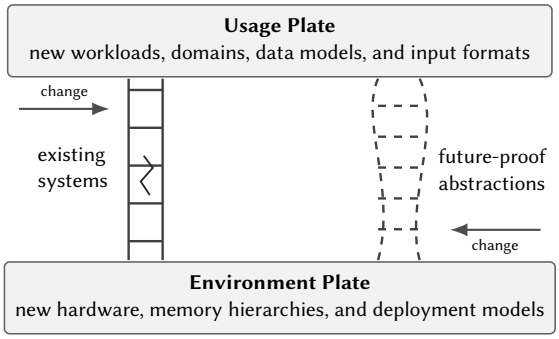

Figure 1: Future-proof data systems must bridge changes in both usage requirements and execution environments.

across systems and increasingly prefer open formats [1, 3, 4, 50], federated setups and modular infrastructure [29, 38] relying on standardized data exchange formats [48, 49]. This weakens the traditional assumptions that one system controls both the data representation and all the computations performed over it.

These trends expose a central limitation of conventional extensibility. Adding a new function, format reader, or an operator may solve an immediate problem, but the added functionality is often *opaque* to the rest of the system. The optimizer may not understand the semantics, the execution engine may not be able to reorganize it, and other systems may not be able to reuse it. Future-proofing therefore requires abstractions that allow new workloads and data representations to be incorporated without turning them into isolated special cases that just bloat the system's internal complexity [22].

2.2 Changing Execution Environment

Even if workloads remained fixed, the execution environment is changing rapidly. Modern systems must exploit tile-based multicore CPUs, accelerators (e.g., GPUs, DPUs, FPGAs, PIM), heterogeneous memory (e.g., HBM, nv-ram), fast interconnects (NVLink, UALink, RDMA, CXL) and increasingly disaggregated resources. In cloud settings, performance depends not only on local operator efficiency, but also on placement, scheduling, materialization , spilling and data movement across resource boundaries.

This volatility makes hard-coded physical implementations increasingly brittle. A specialized operator implementation may be highly efficient on one hardware generation, but difficult to retarget to another. Similarly, a storage or execution strategy tuned for a single-node in-memory system may not naturally translate to a disaggregated or accelerator-based environment. Rather than solely relying on implementations tailored for today's hardware, we should design systems that can abstract away the semantics of the computations from the specifics of the hardware platforms.

Prior work has addressed parts of this problem through code-generation [35, 36], compilation frameworks [11, 16, 23, 34], vectorized execution [53], hardware-specific kernels [17, 24, 25], and adaptive runtimes [14, 40]. Yet, the same tension remains: if the abstraction is too high-level, it hides the state and the data access patterns needed for efficient execution; if it is too low-level, it loses the semantic structure needed for optimization and portability. A future-proof data system must therefore expose enough information for compilers and runtimes to specialize execution, while decoupling the semantic meaning of the computation from the hardware-specific implementation choices [21, 31].

2.3 Limits of local extension points

Specialization and extensibility are the conventional responses to these trends, and both are necessary – but the tight coupling they typically induce limits how independently the individual systems components can evolve while ensuring the end-system remains performant *and* maintainable.

A graph representation optimized for static analytics may not support dynamic updates or pattern matching queries. A query engine that implements file format readers internally must be changed whenever the formats evolve. An operator implementation for one hardware must be rewritten for another. Left unaddressed, this coupling produces systems that are performant locally, but increasingly difficult to maintain and adapt as a whole.

Over the past several years, our research group has tackled this tension by treating *future-proofness as an abstraction-design problem*. In the following three sections we examine how we have addressed it at three specific boundaries across the data processing stack.

3 FUTURE-PROOFING DATA STRUCTURES

The first boundary that we investigated was between workloads and physical data representations. Ideally, applications should not need to deal with customized data structures (or systems) for each workload they need to serve. This would avoid duplicating data and its costly transformations, especially when working with dynamic datasets and application logic that needs to run analytics over it.

This problem was particularly noticeable in graph processing. Graph workloads are highly diverse and used widely across domains: from live recommendation systems [15, 44] to real-time fraud detection [43]. And many of these applications increasingly need to combine different graph computations – from analytics that perform complex, macro-level computations like PageRank to graph traversals or highly specific subgraph searches performing graph pattern matching. These computations are performed on datasets that can be highly dynamic, requiring up to millions of edge insertions per second [44].

Building a system that can efficiently process a diverse set of graph algorithms over dynamic datasets remained an open problem for a long time [30]. It was attributed to the conflicting requirement of the individual computations: analytical algorithms benefit from compact layouts and fast sequential scans over neighborhoods; graph pattern matching needs data to be sorted to enable fast intersections; whereas transactional workloads require support for efficient updates, concurrency control, and the ability to observe a consistent, up-to-date view of the graph. Historically, these conflicting requirements have encouraged the design of specialized systems with customized graph representations, each optimized for a particular workload class or at least two out of the three. For example, most prior work had either focused on static graphs [26, 45] or chose not to support pattern matching when operating on dynamic ones [9, 27].

Instead of focusing on complete workload-specific optimizations, we looked into the key memory access primitives and ranked them

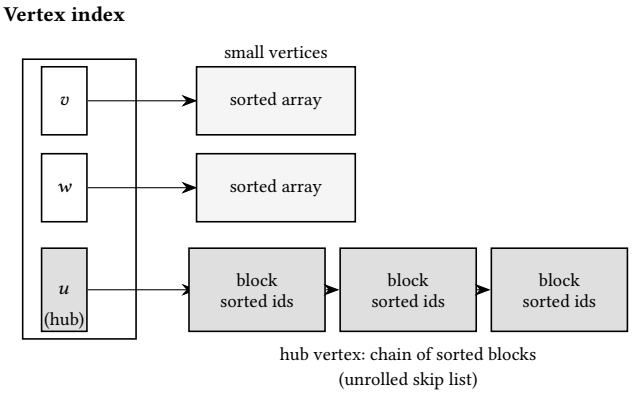

Figure 2: Sortledton's vertex index points to sorted arrays for low-degree vertices and to an unrolled skip list of sorted blocks for high-degree hubs.

by their relative importance. We argue that this way one can preserve flexibility at the level where change (or the differences) are most likely to occur, leading to a more future-proof solution for wider workload mixes.

In the specific case of graph processing the memory patterns that we investigated were: (1) sequential neighborhood access – reading through the list of edges connected to a specific vertex; (2) algorithm-specific property access – reading or writing helper properties (PR scores or BFS distance scores); (3) sequential- and (4) random-vertex access. The core insight was whether to do per-edge or per-vertex optimizations. In contrast to traditional read-only static formats like the Compressed Sparse Row (CSR) that optimized for *both* sequential vertex and neighborhood accesses, we observed that per-edge patterns (neighborhoods or algorithm specific property access) dominate performance and should thus be prioritized over access patterns that occur once per vertex (sequential or random vertex accesses).

Sortledton realizes this through a dynamic adjacency-list design based on sorted, block-based neighborhoods that is better suited for dynamic workloads and significantly simpler to build and maintain (see fig. 2). More specifically:

- To optimize for scan performance, the neighborhoods are broken into large contiguous memory blocks (block-based unrolled skip lists) allowing the CPU to scan the edges almost as fast as a linear memory array.
- To enable fast intersections, the neighborhoods are stored in sorted arrays. This allows for fast binary search merges and quick intersections.
- To support efficient changes, we rely on pointer-linked adjacency lists (with an index). Unlike CSR, which requires big data movements upon insertion, here changes are local. New edges are simply inserted into localized skip-list blocks, allowing up to 5 million updates per second.

Sortledton achieves this without incurring large performance penalty (within 1.22x of static CSR) while maintaining full, real-time update capability with transactional consistency.

In the broader context of future-proof data systems, Sortledton shows that workload evolution can sometimes be absorbed at the data-structure layer by choosing the right access abstraction. Rather than maintaining separate graph representations for analytics, pattern matching, and updates, the system can specialize around stable graph-access primitives. This makes changes in the workload mix less likely to require a new physical representation or another copy of the graph. We next discuss how to broaden the question to the boundary between arbitrary data formats and processing engines.

4 DATA FORMATS

Data management has increasingly moved away from one of the core assumptions of traditional database systems: that data is stored and consumed primarily within a single, vertically integrated system. Classical systems typically relied on proprietary storage formats and highly customized execution engines. Today, data is increasingly stored in open formats, shared through data lakes, and accessed by multiple processing engines. Users expect to move between databases, analytical engines, ML frameworks, and domain-specific tools and libraries. At the same time, many scientific communities continue to rely on specialized formats tailored to their data (e.g., particle physics [7, 13], bioinformatics [8], geo-science).

This shift improves portability and user autonomy, but it also creates a new systems problem. Once we allow data formats and processing engines to evolve independently, every engine that wants to consume a format must implement and maintain a dedicated reader for it. With N systems and M formats, this creates an $N \times M$ implementation and maintenance problem. As a result, new encodings and layouts that offer better performance or compression ratios see limited adoption because integration requires substantial engineering effort. Domain-specific formats struggle to gain traction outside of their communities because mainstream engines do not support them. Existing formats, in turn, risk ossification as extending them requires changes across many systems.

4.1 Need for an abstraction

The underlying issue is again one of abstraction. A processing system should not need to be modified whenever a new encoding, layout, or format-specific optimization is introduced. Conversely, data producers should not be forced to choose only those representations that current engines already support or can parse efficiently. A future-proof boundary between data and systems must therefore *decouple* the format evolution from engine implementation while still preserving direct and efficient access to the underlying data.

At first, these goals appear difficult to reconcile. Hiding format internals suggests introducing an abstraction layer between the data processing system and the data. Yet, direct file access and data sharing are important because they avoid unnecessary data copying. The key question is where the format-specific decoding logic should reside. Placing it inside the engine recreates the $N \times M$ problem; placing it in an external service weakens direct access. The insight behind our proposal is to *bundle the decoder with the data itself*. In other words, datasets should be able to decode themselves.

However, realizing this abstraction without performance penalties and in a safe way raises several systems challenges:

- **portability:** the decoder must be portable across different data processing platforms;
- **security:** the decoder should be sandboxed so that externally supplied decoding logic can be executed safely;
- **performance:** the integration should be efficient not to negate the benefits from the specialized encodings;
- **extensibility:** formulating the decoding should be extensible to support the development of future encodings.

Below, we discuss how we have addressed them with our framework AnyBlox [12].

4.2 AnyBlox

A central design decision in AnyBlox is that it explicitly avoids imposing a new universal data format. Instead, it provides a framework that can be integrated with existing formats, new ones, or even existing datasets (e.g., coming from a scientific community). The abstraction consists of a decoder representation, an interface through which the decoded data is returned to the engine, and the metadata describing the dataset (see fig. 3).

To satisfy the four requirements listed above, AnyBlox uses WebAssembly (Wasm) as a portable and sandboxed execution environment. Wasm is a highly portable, platform-independent instruction set that any engine can execute immediately regardless of its underlying hardware. To decouple the engine's internals from individual datasets, AnyBlox provides a rigid, minimal API through which engines access decoded data, without requiring them to know how that data is compressed or organized. A database engine would simply just call the API functions (e.g., `decode_batch()`) and the embedded Wasm decoder will handle the rest. To ensure minimal performance overhead AnyBlox allows the sandboxed decoder to share memory space directly with the host engine's buffer pool, avoiding expensive data serialization and copying overhead. Finally, by running these decoders in strict Wasm sandboxes the decoder is given access *only* to its own encapsulated file block and the specific memory buffer assigned by the host database.

From the perspective of future-proof data systems, AnyBlox addresses the coupling between data-format evolution and query engine implementation. It allows formats to improve with new layouts, encodings and domain-specific representations without depending on each engine to adopt them manually. This is also useful for open science and reproducibility, because the decoder used to interpret a scientific dataset can remain available with the dataset itself. At the same time, it allows data processing systems to access a broader range of datasets without bloating with a wide collection of bespoke readers. Any system that supports the AnyBlox interface can securely and efficiently read any dataset that provides a compatible decoder.

Looking beyond, open data ecosystems require more than just open file/table formats. They also require abstractions that allow the interpretation of data to evolve with the data itself.

5 EXTENSIBLE QUERY PROCESSING

As a next level of the data processing stack we look at the boundary between computational semantics and physical execution within an execution engine. We argue that now is the right time to reconsider the system abstractions so that data processing engines can easily adapt to different and emerging workloads and can run efficiently in any type of hardware environment, today and in the future. These abstractions must be expressive enough to represent both relational and non-relational logic, structured to support optimization and reasoning, and low-level to expose the information needed for efficient execution on modern hardware.

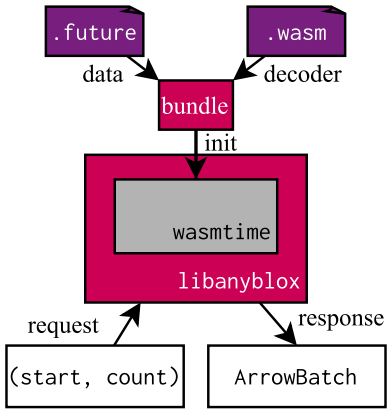

Figure 3: AnyBlox executes a dataset's bundled WebAssembly decoder in a sandboxed runtime and returns decoded Arrow batches through a format-independent interface.

To achieve that we have to solve two relevant problems: First, to represent computations beyond relational algebra without turning them into optimizer-invisible black boxes we propose opening the query engine through extensible intermediate representations. Second, to efficiently map these computations to changing hardware without custom re-implementations for every platform, we introduce declarative sub-operators as a lower-level abstraction that expose state and data access for hardware-conscious optimizations. This way we can separate computational intention from hardware-specific implementation.

5.1 Reasoning beyond relational algebra

Traditional query engines are built around tightly integrated components such as query optimizers, operator implementations, and execution backends. This organization works well when workloads consist of predominantly relational queries over known data types. However, it becomes limiting when applications embed user-defined functions, ML inference or graph computations inside data processing pipelines. When this logic is represented as an opaque function call, the system may execute it, but it cannot optimize it.

Our work addresses this by adopting a compiler-centric design that decomposes the system's functionality. We generalize the concept of query representation and optimization so it can capture both relational and non-relational (e.g., imperative and domain-specific) logic, and enable reasoning and optimization across the traditional system boundaries. More specifically, in our LingoDB design [23] we use *open representations* (OpenIRs) that can also be combined with other intermediate representations from neighboring domains and we implement query optimization as a series of compiler passes.

Our open intermediate representation of relational operators allows arbitrary computations to be embedded within query plans.

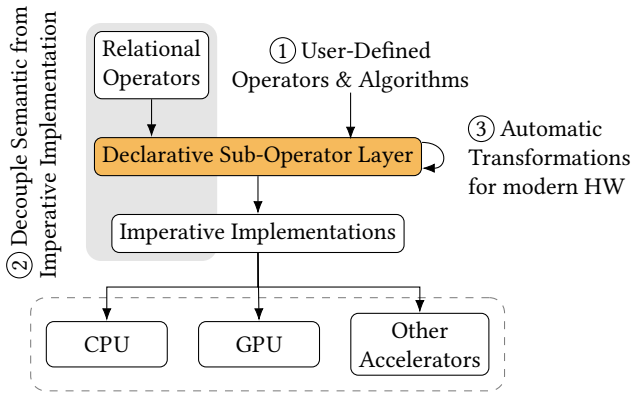

Figure 4: LingoDB's declarative sub-operators layer separates computations over explicit state from their hardware-specific imperative implementations.

For example, a selection operator can be parameterized by a function that consumes a tuple and returns a Boolean value. Because the function itself is represented in the compiler IR, it may contain arbitrary computation, including user-provided logic such as ML inference. For instance, a trained linear-regression model used as a query predicate can be represented in the same IR as the surrounding query, so the same constant-folding and simplification passes that optimize ordinary arithmetic also collapse the inference call into a single inlined inequality — turning what would otherwise be an opaque function call into an expression the optimizer can push down into the scan like any other predicate.

To enable systematic reasoning about such computations, we reformulated query optimization as a collection of extensible compiler transformation passes. Standard compiler optimizations, domain-specific rewrites, and relational transformations can then be interleaved and combined to optimize the full computation. This way we can enable reasoning about non-relational logic and support more sophisticated cross-domain optimizations.

For a unified IR that can represent not only native operators, we propose to use a general but modular compiler (e.g., MLIR) and extend it with domain-specific concepts such as relational operators. Thus, imperative code is natively supported and new domains that often already have support in compiler frameworks can be easily represented as well. This allows for implementing (high-level) optimizations for new domains, and cost-based optimization passes (e.g., operator reordering) can be supported through passes that access the relevant metadata and statistics.

This approach makes the system more future-proof because it separates the evolution of workloads from the fixed structure of the query engine. As new user requirements emerge, such as hybrid relational-vector queries or pipelines that combine SQL with Python-based logic, users and system developers can introduce new representations and transformations without rebuilding the engine around each workload class. The optimizer becomes an extensible framework for reasoning about computation rather than a closed component limited to working with a predefined set of operators and transformations.

5.2 Decoupling semantic logic from efficient implementation on modern hardware

The second challenge is to execute these computations efficiently as hardware evolves. Modern systems must exploit increasing parallelism with CPUs while also adapting to accelerators with different programming models and execution constraints (e.g., GPUs, DPUs, FPGAs or processing-in-memory) and existing or emerging high-bandwidth interconnects (e.g., RDMA, CXL, NVLink, UALink).

If each high-level operator must be re-implemented for every new hardware target, the system quickly becomes difficult to maintain. A future-proof execution engine needs an abstraction that decouples the semantic specification of a computation from the mechanisms used to execute it efficiently on a particular hardware platform (see fig. 4).

We propose declarative sub-operators as such an abstraction [6, 21]. Sub-operators decompose complex operators and dataflows into smaller building blocks with simpler semantics that expose how computations interact with data and state. Unlike high-level relational operators, they make relevant algorithmic structure like state, control-flow and data-structure interactions visible. Unlike imperative low-level code, they remain declarative enough for the compiler and runtime to transform, optimize and re-target. They strike a balance between expressiveness and optimization potential on novel hardware.

The abstraction has three key characteristics:

- First, it makes *state explicit* rather than hiding it inside operator implementations.
- Second, it provides *explicit, though restricted control flow*, making it possible to express a broad class of algorithms and dataflows.
- Third, it remains *declarative*, separating the semantics of the sub-operators from the concrete implementation and execution model.

This allows the same logical computations to be optimized differently for different hardware environments.

Exposing algorithms and state in this way enables the compiler to reason about physical execution choices that are otherwise intertwined in hand-optimized operators. On multicore CPUs, the system would auto-parallelize the code carefully tuning for the cache hierarchy and the critical synchronization points [21]. On GPUs, it may instead focus on reducing control-flow divergence, fine-tuning the parallelization on multiple levels of the cache hierarchy, and organizing memory access to better exploit high-bandwidth memory. In disaggregated environments, explicit state and access patterns could allow the runtime to reason about data placement, movement, materialization and spilling [2, 28].

Future-proofing in this context means that hardware-specific specialization remains possible, but is derived from a reusable representation rather than manually encoded into every operator. As new architectures emerge, the system can introduce new lowering strategies, scheduling policies, or runtime placement decisions beneath the same semantic abstraction. The core engine therefore does not need to be rebuilt for each new platform. Instead, it can adapt by extending the compiler and runtime mechanisms that map the declarative sub-operators to a suitable physical execution.

This principle has also found validation at industrial scale: Microsoft Fabric's CoddSpeed [19] similarly introduces a minimal hardware-agnostic abstraction layer that routes query fragments to various accelerators (e.g., GPUs, FPGAs, and ASICs) without requiring changes to existing SQL workloads or the query optimizer.

Together, the open intermediate representations, treating the optimizer as compiler passes, and the declarative sub-operators pave the way toward query engines that can evolve along the two tectonic plates at once: they incorporate new forms of computation because the optimizer is no longer restricted to relational logic alone, and they adapt to new hardware because implementation is decoupled from semantics.

6 DISCUSSION: DESIGN LESSONS

Across these projects, future-proofing follows a similar design pattern. We identify a boundary where systems are forced into brittle specialization, duplicated functionality, or pairwise integration; we determine which information should remain visible for efficient execution; and we introduce an abstraction that allows independent evolution without hiding the structure needed for optimization.

First, future-proof abstractions should expose stable access properties rather than encode complete workload-specific solutions. Sortledton demonstrates this at the data-structure layer: instead of building customized representations for a subset of analytics, traversals, pattern matching, and transactions, it identifies the core graph-access patterns shared across these workloads and proposes a design focused on the patterns.

Second, future-proof abstractions should move volatile functionality to the side of the boundary where it naturally evolves. AnyBlox demonstrates this for data formats: decoding logic evolves with the data rather than being repeatedly implemented inside each engine.

Third, future-proof abstractions should preserve semantics long enough for optimization, while deferring physical decisions until the execution environment is known. LingoDB's open compilation and optimization stack and the declarative sub-operators demonstrate this for query processing and execution: high-level semantics of the computations remain visible to the compiler passes for optimizations, while hardware-specific specialization is derived through lowerings and runtime decisions.

These lessons also clarify what future-proofness does not mean. It does not mean avoiding specialization, nor does it mean building one system that anticipates every future workload or hardware platform. Rather, it means designing system boundaries so that specialization can be introduced locally, systematically, and without restructuring the system core.

7 PRIOR WORK

The vision of future-proof data systems builds on a long line of work on extensible and modular data management. Extensible database systems such as PostgreSQL [47] and Starburst [32] showed early on that systems should support new data types, access methods and operators. Modern engines like DuckDB [39] continue this tradition through plug-in architectures and native extensions (e.g., pg_vector, duckpgq). These mechanisms are essential for broadening system functionality, but they often rely on system-specific code paths and may expose limited semantics to the optimizer.

User-defined functions and embedded general-purpose languages address the need for more expressive application logic. Procedural extensions (e.g., PL/SQL, PL/pgSQL, SQL/PSM) can be analyzed [18, 41, 46] but are often restrictive and not user-friendly. General-purpose languages (e.g., Python, Java, C++) are expressive but commonly opaque to database systems and notoriously difficult to optimize [20]. Similarly, dataflow frameworks and common runtimes expose useful primitives for large-scale computations [51, 52] and can often optimize across libraries and domain borders [5, 37]. But, computations outside their recognized operators or libraries can still remain black boxes to their optimizer/runtime.

Recent modular systems [29] and standardization efforts including reusable execution engines [38], intermediate representations [48] and in-memory data exchange formats [49] have made it easier to compose components and share functionality across systems. Hardware-conscious systems [14, 53], code generators [35, 36], and accelerator-specific kernels [24, 42] have shown how much performance can be obtained through specialization. Our work is complementary to these efforts. It asks how the abstractions at key system boundaries should be designed so that extensibility, modularity and specialization remain compatible with optimization and long-term evolution and maintenance.

8 CONCLUSION

In this paper we argued that future-proofness is not a property a system achieves once, but rather a system design principle. Each time workloads or hardware changes, the question becomes whether that change can be absorbed by an abstraction that already exists, or if it requires a new one. With the three cases across the system stack we described our approach: locate the boundary where change entails duplication, decide which information has to pass the interface, and move the volatile aspect to the side where it naturally evolves. In the near future, we expect pressure on both plates to grow. For example, cost models would need to reason also across domains, and memory hierarchies and accelerators will continue to diversify the deployment environment. We hope the abstraction-design perspective developed here offers a useful lens as the community confronts the next wave of challenges.

ACKNOWLEDGMENTS

The work summarized in this paper is primarily the result of the creativity, dedication, and hard work of my PhD students and collaborators Michael Jungmair, Mateusz Gienieczko, Per Fuchs, Maximilian Kuschewski, Andre Kohn, and Maximilian Bandle. I am also deeply grateful to all current and former members of my group, including Michalis Georgoulakis, Ferdinand Gruber, Alessandro Fogli, Tobias Goetz, Abdelrahman Adel, Julia Spindler and Calin Pop for creating the broader intellectual environment in which these ideas developed. I am fortunate to be part of a great research community at TUM and to learn from and collaborate with Alfons Kemper, Thomas Neumann, and Viktor Leis. I would also like to thank Peter Boncz, Gustavo Alonso, and Natassa Ailamaki for their encouragement, advice and support throughout my early career as a faculty member. This research was supported in part by the Technical University of Munich, the ERC StG FDS 101164556, and the DFG SPP 2037.

REFERENCES

[1] [n.d.]. Apache Iceberg - Apache Iceberg. https://iceberg.apache.org/

[2] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming Fully Disaggregated Systems. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks (Eds.). ACM, 188–195. https://doi.org/10.1145/3593856.3595889

[3] Apache Parquet. 2024. Parquet. https://parquet.apache.org/. Accessed: 2024-04-30.

[4] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[6] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490. https://doi.org/10.14778/3476249.3476296

[7] Blomer, Jakob, Canal, Philippe, Naumann, Axel, and Piparo, Danilo. 2020. Evolution of the ROOT Tree I/O. *EPJ Web Conf.* 245 (2020), 02030. https://doi.org/10.1051/epjconf/202024502030

[8] Chaoran Chen, Sarah Nadeau, Ivan Topolsky, Niko Beerenwinkel, and Tanja Stadler. 2022. Advancing genomic epidemiology by addressing the bioinformatics bottleneck: Challenges, design principles, and a Swiss example. *Epidemics* 39 (June 2022), 100576. https://doi.org/10.1016/j.epidem.2022.100576 Place: Netherlands.

[9] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[10] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186. https://doi.org/10.14778/3514061.3514065

[11] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2023. The Tensor Data Platform: Towards an AI-centric Database System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://vldb.org/cidrdb/2023/the-tensor-data-platform-towards-an-ai-centric-database-system.html

[12] Mateusz Gienieczko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. 2025. AnyBlox: A Framework for Self-Decoding Datasets. *Proc. VLDB Endow.* 18, 11 (2025), 4017–4031. https://doi.org/10.14778/3749646.3749672

[13] Dan Graur, Ingo Müller, Ghislain Fourny, Gordon T. Watts, Mason Proffitt, and Gustavo Alonso. 2021. Evaluating Query Languages and Systems for High-Energy Physics Data. *CoRR* abs/2104.12615 (2021). arXiv:2104.12615 https://arxiv.org/abs/2104.12615

[14] Tim Gubner and Peter Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *Proc. VLDB Endow.* 16, 4 (2022), 829–841. https://doi.org/10.14778/3574245.3574266

[15] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380. https://doi.org/10.14778/2733004.2733010

[16] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[17] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (2013), 709–720. https://doi.org/10.14778/2536360.2536370

[18] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 723–735. https://doi.org/10.1145/3448016.3457272

[19] Matteo Interlandi, Nicolas Bruno, Brandon Haynes, Carlo Curino, Rathijit Sen, Yinan Li, Kaushik Rajan, Bailu Ding, Lukas M. Maas, Wei Cui, Kevin Gaffney, Mingsheng Hong, Brian Kroth, Sampath Rajendra, Peng Cheng, Surajit Chaudhuri, Johannes Gehrke, Raghu Ramakrishnan, Lidong Zhou, Momin Al-Ghosien, Craig Peeper, Marius Dumitru, Conor Cunningham, Kevin Bocksrocker, Prasanna Sundar, Ron Boskovic, YongChul Kwon, Marko Zivanovic, Runbin Shi, Krystian Sakowski, Denis Lamtsov, Josep Aguilar-Saborit, Krish Srinivasan, Adarsh Kapil, Andrew Putnam, Anudeep Kambapu, Aparna Konduri, Aaron Landy, Aaron Moore, Aaron Pitman, Artem Oks, Ashit Gosalia, Babu Kandimalla, Blake Pelton, Bogdan Crivat, César A. Galindo-Legaria, Chidamber Kulkarni, Jesús Camacho-Rodríguez, Evgeny Babin, Hans Lehnert Marino, Igor Konev, Israel Arroyo, Luis Peñaranda, Mandar Datar, Morten Borup Petersen, Nicholas Simmons, Parminder Poonian, Pravija Danda, Rob Rydberg, Esteban Calvo Vargas, Ronak Bajaj, Sai Sumanth Kammal Shetty, Sharanya Bhat, and Zach Iracheta. 2026. CoddSpeed: Hardware Accelerated Query Processing in Microsoft Fabric. In *Companion of the International Conference on Management of Data, SIGMOD Companion 2026, Bengaluru, India, 31 May 2026 - 5 June 2026*. ACM, 359–372. https://doi.org/10.1145/3788853.3803077

[20] Michael Jungmair, Alexis Engelke, and Jana Giceva. 2024. HiPy: Extracting High-Level Semantics from Python Code for Data Processing. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 736–762. https://doi.org/10.1145/3689737

[21] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474. https://doi.org/10.14778/3611479.3611539

[22] Michael Jungmair and Jana Giceva. 2025. Towards Designing Future-Proof Data Processing Systems. *Proc. VLDB Endow.* 18, 11 (2025), 3988–3995. https://doi.org/10.14778/3749646.3749669

[23] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[24] Marko Kabic, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, 3 (2025), 187:1–187:25. https://doi.org/10.1145/3725324

[25] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (2017), 733–744. https://doi.org/10.14778/3067421.3067423

[26] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2009. Optimistic parallelism requires abstractions. *Commun. ACM* 52, 9 (2009), 89–97. https://doi.org/10.1145/1562164.1562188

[27] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, Arif Merchant and Hakim Weatherspoon (Eds.). USENIX Association, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[28] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27. https://doi.org/10.1145/3698813

[29] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 5–17. https://doi.org/10.1145/3626246.3653368

[30] Dean De Leo and Peter Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. https://doi.org/10.14778/3447689.3447708

[31] Pinghe Li, Tom Kuchler, Marko Kabic, Tobias Stocker, Gustavo Alonso, and Ana Klimovic. 2026. End-to-End Declarative Data Analytics: Co-designing Engines, Interfaces, and Cloud Infrastructure. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. https://vldb.org/cidrdb/2026/end-to-end-declarative-data-analytics-co-designing-engines-interfaces-and-cloud-infrastructure.html

[32] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. 1991. Extensions to Starburst: objects, types, functions, and rules. *Commun. ACM* 34, 10 (Oct. 1991), 94–109. https://doi.org/10.1145/125223.125266

[33] Michael Jungmair. 2026. LingoDB. https://www.lingo-db.com/. Accessed: 2026-07-23.

[34] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 7:1–7:10. https://doi.org/10.1145/3399666.3399911

[35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[36] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html

[37] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[38] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[39] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[40] Bogdan Raducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1231–1242. https://doi.org/10.1145/2463676.2465292

[41] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (2017), 432–444. https://doi.org/10.1145/3186728.3164140

[42] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 2 (2023), 11:1–11:38. https://doi.org/10.1145/3485126

[43] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618. https://doi.org/10.1007/S00778-019-00548-X

[44] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016), 1281–1292. https://doi.org/10.14778/3007263.3007267

[45] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 135–146. https://doi.org/10.1145/2442516.2442530

[46] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 532–543. https://doi.org/10.1109/ICDE.2014.6816679

[47] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *ACM SIGMOD*. ACM, 340–355. https://doi.org/10.1145/16894.16888

[48] Substrait Community. 2026. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io/. Accessed: 2026-07-15.

[49] The Apache Software Foundation. 2024. Apache Arrow | Apache Arrow. https://arrow.apache.org/. Accessed: 2024-05-15.

[50] The Apache Software Foundation. 2024. Apache ORC - High-Performance Columnar Storage for Hadoop. https://orc.apache.org/. Accessed: 2024-04-30.

[51] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 1–14. http://www.usenix.org/events/osdi08/tech/full_papers/yu_y/yu_y.pdf

[52] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[53] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 1349–1350. https://doi.org/10.1109/ICDE.2012.148