Towards Designing Future-Proof Data Processing Systems

Michael Jungmair
jungmair@cit.tum.de
Department of Computer Science, School of CIT
Technical University of Munich

Jana Giceva
jana.giceva@cit.tum.de
Department of Computer Science, School of CIT
Technical University of Munich

ABSTRACT

Data processing systems find themselves crushed between two moving tectonic plates: the usage plate driven by the system's users and their requirements; and the environment plate driven by various technological changes. We argue that the existing status quo of constantly adapting and thus bloating the system's implementation is simply unsustainable in the long run. We further argue that now is the right time to take a step back and establish the foundations of future-proof data processing systems that can easily adapt to different workloads and input formats, and that can run efficiently in any type of environment, today and in the future.

With our paper, we analyze and learn from prior attempts, identify key design principles, and present our vision on how to design such systems.

PVLDB Reference Format:
Michael Jungmair and Jana Giceva. Towards Designing Future-Proof Data Processing Systems. PVLDB, 18(11): 3988 - 3995, 2025.
doi:10.14778/3749646.3749669

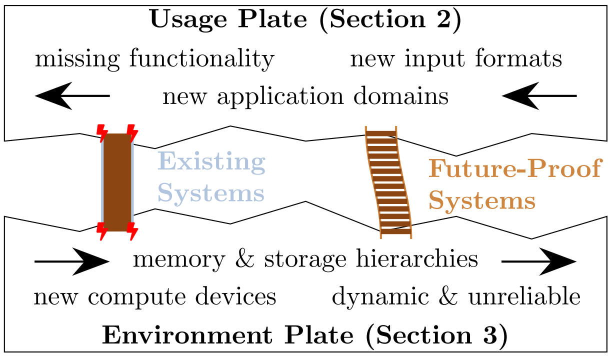

Figure 1: Two *tectonic plates* are moving at the same time. While existing systems cannot easily keep up with the movements like a bridge built out of wood and steel, we envision future-proof systems that can adapt, similar to a rope bridge.

1 INTRODUCTION

Data-intensive processing systems find themselves between *two tectonic plates* (usage & environment) that are continuously moving independently of each other as sketched in Figure 1. Today, many systems already struggle with adapting to one plate due to the increased system complexity. In this paper, we argue that we need to pause and think about the design of future-proof data processing systems that can adapt to the movements of both plates.

The *usage plate* has been in motion for nearly as long as data processing systems have existed. Over the decades, new data models (e.g., object-oriented, graph, NoSQL) have emerged, sparking a drive to develop customized systems (e.g., TigerGraph [66] for graph analytics) before incorporating a suitable support into existing relational systems [64] at the cost of increased code complexity. At the same time workloads and application domains also evolve, for example, with the increased demand to process time-series and spatial data, stream data, or to support machine-learning applications. Most recently, users are increasingly pushing to reduce lock-in effects and expect data processing systems to work on user-provided data, with evolving file (e.g., Parquet [68]) and table (e.g., Delta Lake [4]) formats. While it is now widely being agreed that rebuilding specialized systems every time from scratch is not sustainable [42, 55], it is still not clear how to build one system that is sufficiently flexible and performant to avoid the need for specialized systems at all.

The *environment plate* has also been constantly in movement, but its pace has increased significantly in the last two decades due to several factors: (1) With the end of Dennard's scaling, computing devices have become increasingly parallel and heterogeneous with various accelerators being put in the data path and deployed more widely in hyperscalers. (2) Technological advancements now push towards resource disaggregation relying on low-latency, high-bandwidth networks and most recently cache coherency also across devices (e.g., CXL [16]). The memory hierarchy has also expanded, increasingly blurring the boundaries between memory and storage. (3) Finally, the widespread use of cloud services significantly affected the economic incentives and led to more dynamic environments with elastic scaling, functions as a service, and spot-instances. While there is a large body of research on how to leverage these trends for efficient data processing (e.g., for GPUs [58], FaaS [51]), it is still unclear how to build systems that can run efficiently in any environment with no or only minor changes.

We argue that now is the right time to target both plates at the same time and establish the blueprints of what a future-proof system could look like. However, it is still unclear how one can design systems that are extensible enough to capture new workloads, flexible enough to run in diverse environments, but also performant and maintainable. We analyze existing approaches for adapting to movements in the usage plate (cf. section 2) and the environment plate (cf. section 3), and derive a design for future-proof data processing systems that aims to fulfill all of these criteria.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749669

2 THE USAGE PLATE

We argue that future-proof systems need to be *extensible* to support new user requirements and workloads without needing to rebuild or adapt the core system. Extensible database systems date back to the 1980s, with Postgres designed for extensibility [65] and the Starburst project arguing for it [47]. Today, a multitude of extensions are available for systems like Postgres or DuckDB that add functionality not easily expressible in SQL (e.g., complex business logic), support new domains (e.g., graph processing, vector similarity search, ML), wrap existing libraries (e.g., cryptographic functions), or add support for new file formats (e.g., Parquet, BtrBlocks).

However, despite decades of research in extensible systems, we argue that existing approaches are not sufficient for building future-proof systems. More specifically, we argue that future-proofness requires an extensibility approach that is (1) **user-friendly**, (2) **performant**, and (3) **secure**, raising the following research questions: **(1)** How should one design a user interface for expressing logic that is both high-level and expressive, and allows the system to execute it *efficiently*? This requires *understanding* the provided logic to perform (logical) optimizations, run it in heterogeneous environments, and analyze it from a security standpoint. **(2)** How can the system optimize a combination of (declarative) native operations and user-provided logic? **(3)** How can one avoid a monolithic system by decoupling system components such that they can evolve independently, for example being provided as an extension? **(4)** How can user-provided code that remains a black box be sandboxed and executed efficiently?

We now discuss existing approaches for extensibility to derive insights that help with finding solutions to the research questions.

Native Extensions. Native extensions are frequently used to extend monolithic systems like Postgres (e.g., pg_vector) or DuckDB (e.g., duckpgq). Typically implemented in systems programming languages, they are distributed and loaded as dynamic libraries.

Thus, if the system provides enough extension points, any kind of extension can be implemented, including the reuse of existing libraries. Being written in systems programming languages, native extensions offer similar performance as the core system.

However, using systems programming languages makes native extensions difficult to develop for end-users, and they can also compromise the whole system in case of security issues. It also makes it hard for the database to inline, understand, and optimize the expressed logic and to use it in different execution environments.

Insight 1: Native extensions can be highly performant, versatile, and can make use of existing high-quality libraries.

Deconstructed and Modular Data Processing Systems. Recent years have seen the emergence of deconstructed data processing systems (e.g., Datafusion [42]) and libraries (e.g., Velox [54]) that were well accepted. Instead of building monolithic systems with selected extension points, they opt for a modular system design. This allows for composing customized data processing systems and thus avoiding the need to rebuild systems from scratch for new use-cases. Core ingredients for modular data processing systems are (1) explicit intermediate representations to describe the computations or transformations (e.g., Apache Substrait [15]), and (2) data exchange formats, most prominently Apache Arrow [21].

Modular data processing systems make it particularly easy to exchange core components like the execution engine or the query optimizer [30]. Furthermore, using standardized zero-copy data exchange formats like Apache Arrow allows for reusing existing libraries that use the same format (e.g., implementing support for a file format). Using standardized representations for computations also allows for pushing data filters and transformations down into such external modules [43]. Following the idea of partial evaluation, pushing down computations works even when not all functionality can be supported by the module [50].

Extending such a system with new components typically requires a systems programming language, which makes it difficult for end-users. Furthermore, the security depends on the weakest component, and while modularity enables component replacement, the design itself does not help with adding logical optimizations for new domains or supporting new hardware environments.

Insight 2: Using explicit representations for computations and zero-copy data exchange formats, decouples system components, makes them reusable, and can allow for using existing libraries.

UDFs: Procedural Extensions to SQL. User-defined functions using procedural extensions to SQL (e.g., SQL/PSM, PL/SQL, PL/pgSQL) are typically limited to executing SQL statements, evaluating expressions and control flow, such as if statements and loops.

The limited expressiveness also enables the database system to understand the logic and optimize it to improve performance, for example by inlining [29, 57, 63]. Furthermore, the limited set of language constructs also provides security through language isolation, and the logic is as portable as SQL.

At the same time, the limited expressiveness does not allow for implementing many types of extensions, and combined with commonly outdated syntax, often results in poor adoption by users. Additionally, the embedded interpreter is typically not as fast as (JIT-)compiled programming languages.

Insight 3: Procedural SQL extensions can be understood and securely executed by the system, but offer limited expressiveness.

UDFs: General Programming Languages. Using general-purpose languages (e.g., Python, Java, Rust, C++) for UDFs overcomes the limitations in expressiveness of SQL's procedural extensions.

Thus, most kinds of extensions can be realized this way, including wrapping existing libraries for complex computations, accessing the network, and reading different file and table formats. Using general-purpose programming languages also means users do not need to learn new programming languages, and compiled languages can outperform embedded interpreters for procedural SQL extensions.

However, frequently used high-level programming languages like Python are often challenging to execute efficiently [20] and typically require additional, expensive sandboxing mechanisms for security [18]. Finally, UDFs written in general-purpose programming languages become black boxes for the system, making it difficult to apply optimizations across queries and user-defined logic. It is often nontrivial to support different hardware environments.

Insight 4: High-level languages (e.g., Python) are user-friendly and expressive but not declarative, hard to reason about, difficult to execute securely, and suffer from performance issues.

Common runtimes for data processing. Common runtimes for data processing like Spark [74] or DryadLINQ [73], offer a unified platform for users to express their data processing problems in high-level languages, also using (declarative) functions provided by the system (e.g., filter, map, reduce), and domain-specific libraries that provide some semantics to the system.

Using high-level, general-purpose languages without many constraints makes systems highly user-friendly and flexible. Furthermore, having a unified runtime together with declarative runtime functions allows runtimes to optimize across library or domain borders [53, 74]. For example, loops can be fused across different libraries and user-provided code, leading to a better performance.

However, experience showed that missing high-level operators cause sub-optimal plans [74] and Spark later introduced DataFrames and a query optimizer [5]. Second, everything not directly expressed with runtime functions or runtime libraries remains a black box component written in a high-level language. This is problematic, as high-level languages are often challenging to execute efficiently, provide insufficient security isolation, and are difficult to port for different hardware. To improve execution efficiency, systems like Spark increasingly switch to optimized, native execution engines [8, 62] that only support a limited set of relational operators but cannot run general Spark programs.

Insight 5: Common data processing runtimes can partly reason about computations through runtime primitives despite using high-level programming languages.

Insight 6: Having a unified representation for data processing allows for cross-library/domain optimizations, in addition to high-level optimizations.

3 THE ENVIRONMENT PLATE

Nowadays, data processing systems are not only expected to run on modern, highly parallel CPUs, but also on accelerators with different compute paradigms (e.g., GPUs, processing-in-memory, TPUs, DPUs). Furthermore, environments will become increasingly disaggregated using existing (e.g., RDMA, fast Ethernet) and future (e.g., CXL, NVLink, photonic interconnects) technologies and come with deep and heterogeneous memory and storage hierarchies (e.g., nv-ram, HBM). We argue that future-proof systems should be designed such that the core system does not need to be rebuilt for new hardware configurations. This raises the following question:

How can we decouple the core logic of data processing systems from both user requirements (being extensible), and from the exact hardware environment (being flexible) while remaining performant?

More specifically, this also requires answers to the following challenges: How to design an abstraction for (1) decoupling user requirements from the efficient execution on a variety of compute devices, with different instruction sets and compute paradigms? And how can the core system be decoupled from the complexity of (2) where to place state and optimizing data movements, and (3) how to schedule work and achieve robustness in dynamic and heterogeneous environments.

While these challenges have been recognized before, we argue that now is the right time to treat them as *interconnected.* For example, using accelerators (e.g., GPUs) for data processing effectively not only requires tuning algorithms and generating an optimized binary, but also managing state (e.g., moving working-set to device memory), and optimizing work scheduling across compute devices.

3.1 Targeting heterogeneous compute devices

Various approaches address the challenge of targeting heterogeneous compute devices with differing instruction sets (e.g., SIMD variants) and paradigms (e.g., CPU, GPU, DPU). These include executing pre-compiled hardware-optimized primitives, just-in-time compilation of high-level abstractions, and using declarative sub-operators to tune algorithms for specific hardware.

Pre-compiled primitives. Many approaches use pre-compiled, optimized operator implementations [28, 54], SIMD primitives [67], or GPU-Kernels [35, 37] that are comparatively simple to use and develop, as one can rely on standard development tools and practices. However, because these pre-compiled primitives need to be fine-grained (e.g., operator-at-a-time) to be reusable, inefficiencies (e.g., due to materialization overheads or execution overheads of GPU-Kernels) quickly occur [9, 14, 22]. Furthermore, each primitive either needs to have a custom implementation for each (new) device type (e.g., CPU, NVIDIA-GPUs, AMD-GPUs) [12, 37, 40, 58], or builds on compilers like OpenCL [28, 37], which still requires fine-tuning for each hardware platform [58].

Layered Compilation. In contrast, approaches using layered compilation generate efficient machine code at runtime from high-level operators using multiple layers of abstractions. This is especially used by the machine-learning community to turn high-level tensor programs into efficient code for CPUs, GPUs, and TPUs [46]. While some propose to directly leverage these compilers [24, 27], others build customized stacks for databases [52, 60]. Having multiple layers allows for decoupling user demands (e.g., computations to perform) from hardware capabilities such that higher layers do not need to care about the underlying hardware. Furthermore, multi-layered compilation also allows for building high-level abstractions from low-level, fine-grained primitives, with zero-cost at runtime. Thus, for supporting new hardware, only a few lower layers must be adapted. These advantages come at the price of compilation overheads. The implementation effort can be kept reasonably low by relying on reusable compiler frameworks like MLIR [44], and the surrounding community, which frequently adds hardware support.

Insight 7: Layered compilation allows for decoupling user demands from hardware capabilities and zero-cost abstractions.

Declarative Designs for Sub-Operators. In addition to using a compilation-based approach, there are several proposals for declarative layers below high-level (e.g., relational) operators [25, 33, 56]. In contrast to high-level operators that abstract away from state and data structures, sub-operators typically describe algorithms and their relation to states and data structures. Due to the exposed algorithms and state, it is possible to tune declarative algorithms for different compute paradigms. For example, to efficiently utilize GPUs, it is crucial to minimize control-flow divergence, and optimize data accesses to saturate high-bandwidth memory.

Insight 8: Declarative sub-operators exposing state and state usages help tuning algorithms for different compute paradigms.

3.2 State Management and Dataflow

Reasoning about state and the dataflow is crucial for efficiently utilizing a diverse range of memory and storage options. Over the years, our community has identified the need for exposing state and dataflow, so that specialized runtime components can reason and optimize state placement and data movements.

Page-based interfaces for data access. Such abstractions have existed since the dawn of relational databases in the form of page-based interfaces for data access. For example, buffer managers offer a simple interface for database developers to work with data sizes beyond the available memory and to deal with the complexity of the storage medium. Because the interface is simple and does not include semantics beyond the current access, buffer managers cannot help with optimizing the global dataflow and data movements.

Enhanced interfaces for data access. Newer proposals for interfaces support declaring additional properties. For example, recent work proposes a unified interface for state materialization that allows the underlying runtime to adaptively compress data, switch to other storage mediums, or nodes [41]. It can perform these optimizations, because the interface provides additional metadata that describes the semantics of the materialized data. For example, when the interface includes a hash value for data partitioning (e.g., in a hashtable), the runtime can optimize for the expected data accesses (e.g., a partition at a time) to avoid performance regression. When using such enhanced interfaces, efficient in-memory operators do not need to implement special strategies for utilizing other memory or storage options, for example, in out-of-memory situations.

Explicit State Management for Distribution. In distributed systems, state and dataflow are typically further exposed to abstract away from concrete storage locations, and reason about and optimize data movements. In the simplest case, state is fully exposed for explicit partitioning and transparent shuffling by the underlying system across compute nodes (e.g., MapReduce [17], Spark [5]). More advanced proposals enable annotating additional properties to improve reasoning about dataflow and data movement. For example, tasks in Polaris [2] *logically* describe which cells are required as input and which relations must hold (e.g., left cells and right cells must belong to the same partition for a partitioned hash-join). Exposing state and knowing additional properties allows for reasoning and optimizing for data-flow and resource utilization (e.g., local temporary disk space). The system can reason about the requirements, schedule tasks and make sure that, once required, the state is transferred in time to the compute node, as for example also sketched as a vision for fully-disaggregated data centers [61].

Declarative state allocation. Recent visions for programming fully disaggregated systems propose annotating state allocations over logical memory and storage regions with the required physical properties [3]. For example, annotating high-bandwidth, low-latency, or persistence requirements allows the underlying runtime to late-bind the logical regions on the available physical hardware.

Insight 9: Exposing state and dataflow, annotated with additional properties, allows runtimes to reason about it and optimize state placement and data movements.

3.3 Robust execution in dynamic environments

In dynamic environments, systems must optimize for both performance and *robustness*. Execution should tolerate fluctuating availability of heterogeneous nodes (e.g., spot instances, failures) and data skew. Prior work has proposed several abstractions for increasing robustness, typically by exposing tasks, state, and data-flow.

Task-based parallelism and work stealing. For single-node systems, task-based parallelism combined with work stealing allows for redistributing work in the presence of skew or stragglers. In the database community, this concept has also been introduced as morsel-driven parallelism [45]. The key idea is to use explicit state which, in many cases, is partitioned into thread-local states to avoid synchronization. Tasks are split into smaller subtasks (i.e., morsels) that are scheduled based on data locality and worker availability, with work stealing ensuring adaptive parallelism.

State-centric abstractions for robust distributed systems. For multi-node systems, additional measures are taken to also deal with node failures and *lost state*. For example, Spark uses the abstraction of reliable distributed datasets, where missing state can be recomputed.

Some distributed systems logically separate state from compute, allowing both for reliable execution (as state is not bound to a failing node) and robust execution (allowing for a flexible mapping of compute and storage) [2]. Additionally, state-centric designs also support on-demand state-separation by evacuating nodes in failure scenarios (e.g., spot instances being notified to be retracted) [71].

Insight 10: State-centric abstractions enable robust task execution in dynamic environments.

4 DESIGN PROPOSAL

Building on sections 2 and 3, we present our vision for building future-proof data processing systems that are extensible, flexible, and performant. Figure 2 gives an overview of the proposal.

4.1 Extensibility

① *Language Support.* In addition to working with query languages like SQL, users should be able to extend the system with a user-friendly and expressive, yet performant, secure, and potentially portable language. To enable optimizations and increase portability, the language must include declarative elements, either through its design (Insight 3) or via declarative libraries that should primarily be used (Insight 5). Ideally, an intermediate representation can be produced from the language that captures declarative and imperative elements.

One could develop a new domain-specific language along these requirements, but gaining sufficient adoption for this language could become difficult. Instead, we propose to take a popular high-level, expressive programming language like Python (Insight 4), and supplement it with declarative libraries. An analysis tool then *extracts the high-level semantics* from the language and produces an IR that can be understood by the system [32] and contains high-level operators (e.g., from pandas operations), declarative sub-operators (e.g., produced from declarative libraries), and imperative operations. For a dynamic language like Python, not everything can be fully analyzed, but remaining black boxes of Python code can be embedded in the IR, and later executed by a sandboxed interpreter.

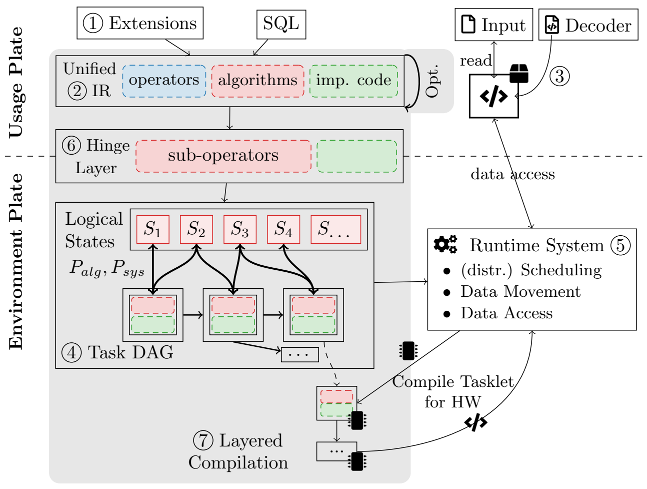

Figure 2: Our vision for building future-proof data processing systems. For the user plate, both SQL and ① Extensions produce a ② unified IR enabling powerful optimizations. For the environment plate, a ④ directed task graph explicitly describing the relation between tasks and logical state, allows for declaratively delegating many decisions to a ⑤ runtime supporting highly complex and dynamic environments. Finally, a ⑥ hinge layer containing declarative sub-operators and ⑦ layered compilation connects both plates.

② *Representing and Optimizing native and user-provided logic.* We argue that future-proof systems need to reason about both native and user-provided logic and operators of new domains.

For a *unified IR* that can represent not only native operators, we propose to use a general but modular compiler IR (e.g., MLIR [44]) and extend it with domain-specific concepts such as relational operators, in contrast to using a domain-specific representation (e.g., tree of operator objects). Thus, imperative code is natively supported, and new domains (e.g., ML) that often already have support in compiler frameworks can be easily represented as well. Similarly, this approach allows for implementing (high-level) optimizations for new domains and imperative code as unconstrained compiler passes on a compiler IR, ideally being able to reuse existing implementations. At the same time, cost-based optimization passes (e.g., operator reordering) can be supported through passes that access the relevant statistics and metadata and annotate operations [34].

The exact semantics of operators is determined by lambda functions that are represented in the same modular IR. For example, a selection operator takes a lambda function that consumes a tuple and produces a boolean result. This function can contain arbitrary IR operations, and, thus, also represent user-provided logic like inference for machine-learning models (e.g., decision trees). Cross-domain optimization then becomes feasible by *interleaving* different compiler passes, that e.g., optimize the inference by interleaving constant propagation with simplification passes [36].

For new user requirements (e.g., hybrid relational-vector queries, or machine-learning), users can extend the system by writing code in a high-level language that is translated into the unified IR and is optimized by a combination of native and user-provided passes. Due to relying on modular compiler technology, deeper integrations in the system can be achieved without major rewrites. Instead, new IR operations and optimization passes can be defined (e.g., in the form of a new MLIR dialect) and integrated into the system (e.g., by a third party for optimized hybrid relational vector queries [48]).

③ *Runtime Extensibility for Input Formats.* To support future file formats, new compressions and encodings inside existing formats, table formats, and storage services on the fly, in addition to natively supported formats, we propose that users provide the necessary logic for accessing the data (e.g., embedded inside file formats).

However, data access logic will typically be written in systems programming languages (Insight 1) to (1) use efficient hardware primitives (e.g., SIMD) and (2) reuse existing libraries (e.g., Parquet reader, Delta Lake library). To be able to securely execute them, future systems need to provide isolation mechanisms. Compiler-based virtualization techniques (e.g., WebAssembly [26]) seem promising due to their standardized format and portability.

For the interface to such extensions, future-proof systems should rely on a standardized and efficient in-memory data format such as Apache Arrow (Insight 2) for accessing (pre-filtered) data originally stored in other formats. Materialization overheads can be avoided by only converting data on-demand in cache-sized data chunks (e.g., as requested by a scan). Efficient compression techniques can still be leveraged by executing filters on the native format and only converting *filtered* data to the standardized format. However, pushing predicates down into extensions will require (1) standardized predicate representations and (2) negotiation over which filtering steps are pushed down. While Substrait [15] could be a candidate for (1), (2) could be handled by having the decoder return any filtering steps it cannot perform, following the idea of partial evaluation [50].

With our proposal, users can extend the system to process new input formats (e.g., FastLanes [1], or the ROOT format used by physicists [13]), or support new compression techniques, by providing a decoder, e.g., written in Rust and compiled to WebAssembly. The system can then process files at near-native performance, without adapting the system or compromising security.

4.2 Complex and dynamic environments

To support increasingly dynamic and complex environments, we propose a ④ state-centric abstraction (Insights 9, 10) for describing both compute tasks and the required (intermediate) state.

More specifically, we propose representing computations as a (1) directed, acyclic graph of tasks that (2) interact with a set of *logical states*. Each task contains a so-called *tasklet* that describes the computations performed for each part into which the task can be split. For example, a task could be scanning a table and do further data processing on top. In this case, the corresponding tasklet would then describe (e.g., in the form of an IR) the necessary computations for a single data chunk of the scanned table. Furthermore, each task describes the relation of tasklets and logical states in two dimensions. First, algorithmic properties (P_{alg}) describe the algorithmic constraints for scheduling tasklets and their inputs. For example, for a task implementing a partitioned hash-join, these constraints should specify that each tasklet needs to consume part of each input relation, but only full partitions, and the partitions must match. Similarly, further constraints can specify which sets of tasklet instances are allowed to be executed concurrently without additional synchronization, or access shared state. The second dimension describes desired properties P_{sys} from a system's point of view. Such properties could include whether the task requires low-latency or high-bandwidth access to some logical state, the required persistence guarantees, or the confidentiality of data. Making these constraints on the relations of tasks and states explicit allows the system to ⑤ reason about task placement, state management, dataflow and robustness, and optimize for it at runtime.

Let's assume that a *new data storage technology* should be integrated that offers new performance characteristics (e.g., higher bandwidth with HBM), new guarantees (e.g., durability with non-volatile RAM) or properties (e.g., shared memory across CPU and accelerators), but also limitations (e.g., limited capacity, write amplification, higher latency). The introduced abstractions capture the algorithmic and physical requirements necessary for effectively delegating placement decisions to a runtime system, avoiding the need to adapt the core system from to new storage technologies.

Similarly, *new data movement* technologies offer higher bandwidths [10] or lower latencies (e.g., photonic interconnects [49]), new guarantees (e.g., cache coherency with CXL or NVLink), and thus may require adapting the system's strategy for scheduling work and placing data for increasing efficiency. The task DAG capturing both abstract computations and their interaction with state enables a dedicated runtime system to reason about different options of moving data and/or compute, depending on access patterns, compute capabilities, the task's operational intensity [70], and the capabilities of interconnects. This avoids the need to adapt the core system, especially if the runtime component can be (largely) reused.

4.3 Connecting both plates

To bridge both tectonic plates and support evolving workloads in changing environments, we rely on two main ideas: (1) a declarative yet flexible abstraction layer that acts as a "hinge" between the unified IR (usage plate) and the annotated task graph (environment plate), and (2) layered compilation for compiling native and user-defined code to machine code for heterogeneous devices.

⑥ *"Hinge" Layer.* The proposed hinge layer fully exposes state and data-structures and consists of *declarative* sub-operators that perform computations and interact with the explicit state. Furthermore, most sub-operators can be customized through attached lambda-functions that determine their exact behavior (e.g., for sub-operators like `map` or `reduce`), using arbitrary operations from the unified IR. This allows capturing any compute (in the worst-case, as black box inside one sub-operator) and flexibly building custom algorithms. At the same time, the declarative design of the sub-operator still allows for rewrites and optimizations even when the embedded logic is not fully understood, and contains enough information (especially regarding the interaction with states) to derive further properties (P_{alg}, P_{sys}), which users cannot be expected to annotate manually. Furthermore, if sub-operators are used for describing tasklets, the computation can be adapted for different compute paradigms (Insight 8), for example, when a set of tasklets is scheduled to run on a GPU. Our recent design for declarative sub-operator layer [33] could serve as a basis for the hinge layer, and ongoing work indicates suitability also for targeting GPUs.

⑦ *Multi-layered Compilation.* A multi-layered compilation approach allows systems to go from high-level, device-generic abstractions to low-level device-specific primitives, with multiple layers in between. This enables compiling a unified IR, as discussed in subsection 4.1, and imperative operations e.g., embedded in sub-operators into efficient machine code for different devices, also enabling efficient execution of extension-provided code. It also eliminates the need for high-level, device-specific operators and further avoids overheads that quickly occur when composing high-level operators from lower-level, compiled primitives (Insight 7). Compilation for a specific device is only started after the runtime has placed tasklets on this device type (e.g., using runtime information and statistics).

Let's assume that the system is required to support new *compute technologies* (e.g., processing-in-memory (PIM), TPUs, NPUs, Xilinx's AiEngines). The properties of sub-operators to perform semantically equivalent rewrites combined with the ability to reason about compute, state and hardware capabilities allow for easier adaptation to different compute paradigms (e.g., minimize control-flow divergence in SIMT architectures). Thanks to layered compilation, generating efficient binaries for new devices only requires exchanging the lowest layers that are often already available when relying on compiler frameworks like MLIR (e.g., for PIM [39] or AIEngines [72]). Finally, further properties that can be derived from sub-operators (e.g., idempotency of sub-operators) can be useful for dealing with new technologies with new types of failures [6].

4.4 Open Research Challenges

With this paper, we aim to spark a discussion on the research questions involved in building future-proof data processing systems, and propose a possible blueprint for such a system design. However, many aspects remain open challenges, often at the boundaries with neighboring domains like systems and programming languages.

(1) User Interface for End-Users The more ambitious approach of designing a new, suitable domain-specific language (i.e., PL/SQL in 2025) could be superior, if the necessary adoption can be achieved.

(2) Generic Runtime Component There is a huge potential for a runtime system usable across disciplines, onto which many decisions such as memory allocation (e.g., as already started with [3, 31]), scheduling and data movement are carefully delegated.

(3) Verifying Extensible Compilers Using extensible compiler frameworks for optimizing and compiling user-provided code requires that transformation steps do not introduce security issues. While research on validation and verification for extensible frameworks has already started [7, 11, 19, 69], it requires further investments, also from our community (e.g., for query optimization).

(4) Compilation Overheads While techniques for low-latency compilation are well-known for CPUs [23, 38, 59], further research is required for hardware platforms like GPUs and FPGAs.

ACKNOWLEDGMENTS

Google supported this work with a PhD Fellowship. Funded/Co-funded by the European Union (ERC, FDS, 101164556). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144.

[2] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[3] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming Fully Disaggregated Systems. In *HotOS*. ACM, 188–195.

[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.

[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 *SIGMOD Conference*. ACM, 1383–1394.

[6] Gal Assa, Michal Friedman, and Ori Lahav. 2024. A Programming Model for Disaggregated Memory over CXL. *CoRR* abs/2407.16300 (2024).

[7] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. SMT-Based Translation Validation for Machine Learning Compiler. In *CAV (2) (Lecture Notes in Computer Science)*, Vol. 13372. Springer, 386–407.

[8] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD Conference*. ACM, 2326–2339.

[9] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. 2023. Evaluating SIMD Compiler-Intrinsics for Database Systems. In *VLDB Workshops (CEUR Workshop Proceedings)*, Vol. 3462. CEUR-WS.org.

[10] André Berthold, Constantin Fürst, Antonia Obersteiner, Lennart Schmidt, Dirk Habich, Wolfgang Lehner, and Horst Schirmeier. 2024. Demystifying Intel Data Streaming Accelerator for In-Memory Data Processing. In *DIMES@SOSP*. ACM, 9–16.

[11] Siddharth Bhat, Alex C. Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. In *ITP (LIPIcs)*, Vol. 309. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:20.

[12] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209.

[13] CERN. 1995. CERN ROOT Format. https://root.cern/manual/root_files/#storing-columnar-data-in-a-root-file-and-reading-it-back. Accessed: 2025-05-08.

[14] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556.

[15] Substrait Community. 2025. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io/ Accessed: 2025-02-11.

[16] Compute Express Link Consortium. 2025. Compute Express Link (CXL). https://computeexpresslink.org/ Accessed: 2025-02-27.

[17] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. USENIX Association, 137–150.

[18] Jake Edge. 2013. The failure of pysandbox. *LWN.net* (2013). https://lwn.net/Articles/574215/ Accessed: 2025-07-08.

[19] Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proceedings of the ACM on Programming Languages* 9, PLDI (June 2025), 206:1–206:25. https://doi.org/10.1145/3729309

[20] Yannis Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *ICDE*. IEEE, 3593–3598.

[21] Apache Software Foundation. 2025. Apache Arrow: The universal columnar format and multi-language toolbox for fast data interchange and in-memory analytics. https://arrow.apache.org/ Accessed: 2025-02-11.

[22] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD Conference*. ACM, 1603–1618.

[23] Henning Funke, Jan Mühlig, and Jens Teubner. 2022. Low-latency query compilation. *VLDB J.* 31, 6 (2022), 1171–1184.

[24] 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 *CIDR*. www.cidrdb.org.

[25] Tim Gubner and Peter A. 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.

[26] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 185–200.

[27] 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.

[28] 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.

[29] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *SIGMOD Conference*. ACM, 723–735.

[30] InfluxData Inc. 2023. InfluxDB — open source time series, metrics, and analytics database. https://influxdata.com/ Accessed: 2025-02-28.

[31] Intel. 2024. Unified Memory Framework. https://oneapi-src.github.io/unified-memory-framework/introduction.html. Accessed: 2025-05-28.

[32] 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.

[33] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474.

[34] 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.

[35] Marko Kabić, 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

[36] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *CIDR*. www.cidrdb.org.

[37] 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.

[38] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[39] Asif Ali Khan, Hamid Farzaneh, Karl Friedrich Alexander Friebel, Clément Fournier, Lorenzo Chelini, and Jerónimo Castrillón. 2024. CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms. In *ASPLOS (4)*. ACM, 31–46.

[40] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD Conference*. ACM, 555–569.

[41] 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.

[42] 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 *SIGMOD Conference Companion*. ACM, 5–17.

[43] LanceDB. 2025. Filter Push-Down. https://lancedb.github.io/lance/read_and_write.html#filter-push-down Accessed: 2025-02-28.

[44] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. IEEE, 2–14.

[45] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[46] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 708–727.

[47] Guy M. Lohman, Bruce G. Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. 1991. Extensions to Starburst: Objects, Types, Functions, and Rules. *Commun. ACM* 34, 10 (1991), 94–109.

[48] Rui Ma, Kai Zhang, Zhenying He, Yinan Jing, X. Wang, and Zhenqiang Chen. 2025. CHASE: A Native Relational Database for Hybrid Queries on Structured and Unstructured Data. (01 2025). https://doi.org/10.48550/arXiv.2501.05006

[49] George Michelogiannakis, Yehia Arafa, Brandon Cook, Liang Yuan Dai, Abdel-Hameed A. Badawy, Madeleine Glick, Yuyang Wang, Keren Bergman, and John Shalf. 2023. Efficient Intra-Rack Resource Disaggregation for HPC Using Co-Packaged DWDM Photonics. In *CLUSTER*. IEEE, 158–172.

[50] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow.* 17, 4 (2023), 877–890.

[51] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference.* ACM, 115–130.

[52] 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 *DaMoN.* ACM, 7:1–7:10.

[53] 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.

[54] 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.

[55] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satyanarayana R. Valluri, Mohamed Zaït, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (2023), 2679–2685.

[56] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718.

[57] 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.

[58] 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.

[59] Tobias Schwarz, Tobias Kamm, and Alexis Engelke. 2025. TPDE: A Fast Adaptable Compiler Back-End Framework. *CoRR* abs/2505.22610 (2025).

[60] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD Conference.* ACM, 1907–1922.

[61] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. 2022. Towards a fully disaggregated and programmable data center. In *APSys.* ACM, 18–28.

[62] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E. Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era. In *VLDB Workshops (CEUR Workshop Proceedings)*, Vol. 3462. CEUR-WS.org.

[63] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE.* IEEE Computer Society, 532–543.

[64] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around.. *SIGMOD Rec.* 53, 2 (2024), 21–37.

[65] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD Conference.* ACM Press, 340–355.

[66] TigerGraph. 2024. TigerGraph. https://www.tigergraph.com/ Accessed: 2024.

[67] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. 2020. Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. In *CIDR.* www.cidrdb.org.

[68] Deepak Vohra. 2016. *Apache Parquet.* Apress, Berkeley, CA, 325–335. https://doi.org/10.1007/978-1-4842-2199-0_8

[69] Yanzhao Wang, Fei Xie, Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2024. A Systematic Translation Validation Framework for MLIR-Based Compilers. *Int. J. Softw. Eng. Knowl. Eng.* 34, 10 (2024), 1621–1640.

[70] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[71] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979.

[72] Xilinx. 2020. MLIR-AIE: MLIR-based Compiler Infrastructure for AI Engines. https://github.com/Xilinx/mlir-aie. Accessed: 2025-05-19.

[73] 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 *OSDI.* USENIX Association, 1–14.

[74] 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.