ESP IJAST
ESP International Journal of Advancements in Science & Technology
ISSN: 2583-9233 / Volume 4 Issue 1 March 2026
Paper Id: IJAST-V4I1P111 / Page No: 81-90

Original Article

Data System Languages: An Expressive Framework for Distributed Heterogeneous Data Analytics

Velu Kaliappan
EY, Boston, United States of America.

Received Date: 28 February 2026 Revised Date: 09 March 2026 Accepted Date: 26 March 2026

Abstract: *This paper presents a new programming language designed for the evolving landscape of data analytics, where the increasing complexity of data pipelines demands robust support for varied data types and scalable operations. It introduces a novel language structure that facilitates data parallel processing across both data streams and data frames within a distributed computing environment. The paper details the language's formal definition and its integration with modern compilation frameworks for efficient execution. By enabling flexible expression and deployment for diverse analytical tasks, this system offers an advanced paradigm for handling the challenges of large-scale, multi-modal data analysis.*

Keywords: *Domain-Specific Language (DSL), Arc-Lang, Data Analytics, Intermediate Representation (IR), MLIR, Compiler Optimization, Rust Code Generation, Stream Processing, Tensor Computation, Graph Analytics, Distributed Systems, Type-Safe Programming, Dataflow Execution, IR Dialects, Runtime Systems*.

I. INTRODUCTION

The exponential increase in data generation—fueled by pervasive digitalization, ubiquitous sensors, and interconnected platforms—has significantly transformed the landscape of data processing. Modern applications in domains such as social media, sensor telemetry, financial systems, and climate modeling routinely handle high-volume, high-velocity, and high-variety datasets. These developments demand robust, flexible computing frameworks capable of processing complex workloads efficiently across diverse infrastructures, including personal devices, edge computing platforms, and cloud-scale clusters. In response to these computational challenges, the field has increasingly embraced *distributed shared-nothing architectures*, wherein large datasets are partitioned and processed in parallel across multiple independent nodes. While these architectures provide scalability and fault isolation, they also introduce substantial programming complexity. Developers must account for coordination mechanisms, resilience to partial failures, data consistency models, and performance tuning, all of which complicate application
development and maintenance.

To abstract these difficulties, a variety of high-level Domain-Specific Languages (DSLs) have emerged. These languages enable developers to express complex data transformations using intuitive constructs aligned with specific problem domains. For instance, SQL provides declarative querying over structured data; Apache Beam [1] supports unified batch and stream processing; Keras [2] simplifies the construc- tion of deep learning models; and Cypher [3] offers expressive pattern-matching for property graphs. Despite their utility, these DSLs often operate in isolation and lack interoperability, which becomes problematic for applications that span multiple data types or computational paradigms.

This siloed approach gives rise to multiple issues. First, integrating different DSLs necessitates expensive data serialization, transformation, and movement across system boundaries. Second, in-consistencies in runtime behaviors—such as varying fault tolerance guarantees, availability models, or security assumptions—can introduce subtle errors. Third, developers are burdened with learning disparate syntaxes and operational semantics, increasing the learning curve and reducing productivity. To address these limitations, we propose Arc-Lang—a novel DSL for unified and scalable data analytics. Arc-Lang is designed to express computations over diverse high-level data types such as streams, tensors, relational frames, and graphs, within a coherent and type-safe programming environment. At its core, Arc-Lang leverages the Multi-Level Intermediate Representation (MLIR) framework to enable modular compilation, domain-specific optimization, and interoperability. The resulting intermediate code is compiled into efficient, type-safe Rust programs that can be deployed on distributed runtimes.

The key contributions of this paper are as follows:

- We introduce the Arc-Lang language and its syntax for data analytics, supporting user-defined behavior and modular composition across data types.

This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/2.0/)

- We present the design of the Arc-MLIR dialect, enabling extensibility and optimization through standard and custom compiler passes.
- We describe the compilation flow that translates Arc-Lang into Rust via MLIR, and how the resulting code integrates with a distributed runtime system.
- We evaluate Arc-Lang against contemporary DSLs across expressiveness, performance potential, and architectural modularity.

The remainder of this paper is organized as follows: Section 2 reviews related DSLs and intermediate representations. Section 3 details the Arc-Lang language and features. Section 4 explains the MLIR- based compilation strategy and runtime considerations. Section 5 provides evaluation metrics and experimental results. Section 6 discusses trade-offs and extensibility. We conclude in Section 7 and outline future directions in Section 8.

II. RELATED WORK

The increasing diversity and volume of data have motivated the emergence of various Domain-Specific Languages (DSLs) and frameworks aimed at simplifying large-scale data processing. These systems have attempted to abstract away the complexity of distributed computing, while enabling expressive and optimized computation over different data types and processing models.

A. DSLs for Structured and Stream Data

Relational query languages like SQL remain foundational in data analytics. Modern distributed variants such as Apache Hive [4] and Spark SQL [5] extend traditional SQL for large-scale batch processing. Apache Beam [1] and Flink [6] further generalize this to unify batch and streaming paradigms through high-level abstractions.

Stream processing frameworks such as Apache Storm [7], Kafka Streams [8], and Google Cloud Dataflow [9] have introduced scalable APIs for handling real-time data flows. While effective in their domain, they lack built-in interoperability and often require external coordination when integrating with systems that process other data modalities.

B. Machine Learning and Tensor DSLs

The machine learning ecosystem includes powerful DSLs such as TensorFlow [10], PyTorch [11], and Keras [2]. These languages provide imperative and declarative interfaces to define and optimize tensor operations. While they offer abstraction and efficiency, they are often limited to numerical and neural network-centric computations.

Systems like TVM [12] and XLA [13] attempt to bridge language expression with compiler optimiza- tion by compiling tensor programs into optimized backend code. However, these approaches focus largely on hardware acceleration rather than generalized distributed analytics.

C. Graph DSLs and Analytics Frameworks

In the domain of graph analytics, languages like Cypher [3], Gremlin [14], and GSQL [15] support expressive traversal and pattern-matching on graph data. Distributed frameworks such as Pregel [16], GraphX [17], and PowerGraph [18] facilitate parallel graph computation. However, graph-specific DSLs often operate in isolation, lacking integration with tensor, stream, or tabular data.

D. Intermediate Representations and Compilation Frameworks

A critical innovation in language design is the development of extensible Intermediate Representations (IRs) like MLIR [19] and Weld [20]. MLIR enables the construction of custom dialects, allowing multiple DSLs to share a common infrastructure for parsing, transformation, and code generation. Arc-Lang builds upon this capability to support typed data transformations and compilation into efficient Rust code.

LLVM [21], upon which MLIR is based, has long served as a backend for language compilers, while Halide [22] and Terra [23] demonstrated domain-specific compilation strategies. However, these systems often target performance over extensibility and language interoperability.

E. Distributed DSLs and Unified Models

Several DSLs and IRs attempt to unify distributed data processing. Systems like DryadLINQ [24] and Naiad [25] proposed data-parallel abstractions with custom execution engines. More recently, frameworks such as Ray [26], Dask [27], and Mars [28] offer Python-native DSLs for parallel computing. However, their IRs are not always explicitly extensible or tailored for compilation into type-safe lower- level languages like Rust.

F. Positioning Arc-Lang

Arc-Lang differentiates itself by offering a type-safe, extensible DSL that can model and process multiple data types—streams, graphs, tensors, and relational frames—under a unified MLIR-based compilation strategy. By compiling into Rust, Arc-Lang benefits from strong safety guarantees and efficient execution in distributed environments. Moreover, the modular MLIR dialect design enables the incorporation of domain-specific transformations without compromising portability.

Table 1: Comparison of Arc-Lang with Representative DSLs and IRs

System	Data Types	IR Extensibility	Rust Output	Target Domain
Apache Beam	Streams	No	No	Dataflow
Spark SQL TVM	Tables	No Partial	No No	Batch & Stream ML
Cypher Weld	Tensors	No Yes Yes	No No	Compiler Graph
MLIR	Graphs		Yes (via Arc-	Query DSL Backend
	Generic Any	Yes	Lang) Yes	IR Framework
Arc-Lang				
	Streams, Graphs, Ten- sors			General Analytics

As shown in Table 1, Arc-Lang combines the strengths of multiple existing systems while addressing their gaps through unified typing, extensibility, and efficient code generation.

III. METHODOLOGY

Arc-Lang is architected as a Domain-Specific Language (DSL) that enables declarative and efficient expression of data analytics computations across diverse data types. The system is composed of three primary layers: (1) a high-level DSL frontend, (2) an MLIR-based intermediate representation layer for optimization and transformation, and (3) a backend that emits Rust code for distributed deployment. This section details each component, the interaction between them, and the underlying architectural principles.

A. Design Objectives

The methodological framework behind Arc-Lang was driven by the following core design objectives:

- Data Abstraction: Provide unified support for high-level data structures such as streams, tensors, graphs, and frames within a single language model.
- Extensible Intermediate Representation: Leverage MLIR to enable reusable dialects and transformations across multiple domains.
- Distributed Execution Readiness: Compile Arc-Lang into type-safe Rust code that integrates seamlessly with distributed runtime systems.
- Optimization-Oriented Architecture: Facilitate performance-enhancing transformations such as operator fusion, common subexpression elimination, and constant folding.

B. System Architecture

The high-level system architecture is illustrated in Fig. 1. Arc-Lang source code is parsed into an Abstract Syntax Tree (AST), transformed into MLIR dialects, optimized, and compiled into Rust code for deployment.

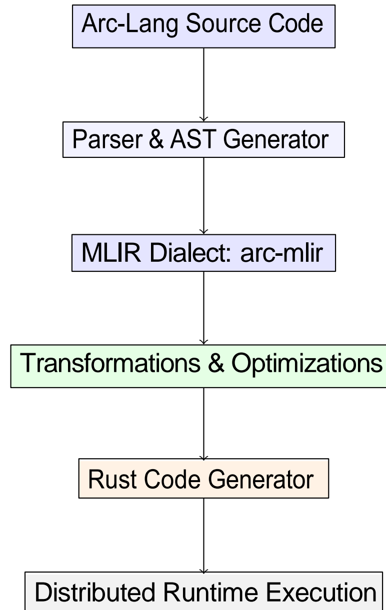

Figure 1: Arc-Lang compilation and execution pipeline.

C. Arc-Lang DSL and Typing System

Arc-Lang allows the definition of high-level computations over structured data types. Its core abstrac- tions include:

- Streams: Represent time-ordered sequences of events.
- Frames: Tabular structures similar to dataframes in Pandas.
- Graphs: Vertex and edge data representations for analytics.
- Tensors: Multi-dimensional arrays for numerical computation.

All types in Arc-Lang are strongly typed, and custom data types such as enums and structs are supported. The type system ensures semantic correctness and supports polymorphic operations.

D. MLIR Integration and Arc Dialect

Arc-Lang's compilation process involves translation into a custom MLIR dialect called arc-mlir. This dialect introduces specialized operations and types corresponding to Arc-Lang constructs. It builds upon standard MLIR dialects such as scf, arith, and std, and supports:

- arc.struct, arc.enum – user-defined structured types.
- arc.stream<T> – stream abstraction for event data.
- Custom arithmetic operations aligned with Rust's type semantics.

MLIR allows modular transformation passes, including canonicalization, constant folding, and custom fusion passes specific to Arc-Lang semantics.

E. Optimization Pipeline

Several optimization passes are implemented on the arc-mlir IR:

- Common Subexpression Elimination (CSE): Identifies and merges duplicate computations.
- Operator Fusion: Combines adjacent operations to reduce memory overhead.
- Dead Code Elimination: Removes unused definitions.
- Type-Specific Rewrite Rules: Optimizes based on user-defined data types.

These transformations improve performance and are structured as composable MLIR passes that can be selected via CLI in the `arc-mlir` tool.

F. Rust Code Generation and Runtime Mapping

Following optimization, the IR is lowered into a Rust-compatible dialect that preserves typing, control flow, and memory safety. The resulting Rust code is then compiled using rustc, optionally incorporat- ing concurrency libraries (e.g., tokio, rayon) and Arc-Lang runtime macros to manage borrowing, ownership, and distributed state management.

G. Runtime Execution Model

The Arc-Runtime library abstracts reference counting, synchronization, and stream consumption, enabling safe concurrent execution. The runtime is deployed across nodes, each responsible for partitioned data processing. Fault tolerance is achieved through snapshotting and idempotent event replay. Load balancing and scheduling are performed using a central coordinator or a decentralized gossip protocol depending on deployment mode.

IV. IMPLEMENTATION

The implementation of Arc-Lang encompasses the development of a compiler toolchain, a set of custom MLIR dialects, transformation passes, and an execution backend targeting Rust. This section describes the major components and tools used to realize Arc-Lang from its high-level syntax down to its executable distributed form.

A. Compiler Frontend

The Arc-Lang frontend is responsible for parsing user-written source code into an Abstract Syntax Tree (AST). The language features a statically-typed functional syntax with support for high-level constructs like pattern matching, structural typing, and first-class streams. The parser and type-checker are implemented in Rust using libraries such as lalrpop for grammar parsing and logos for lexical analysis.

The output of the frontend is a typed intermediate representation (IR), which is then lowered into the Arc-specific MLIR dialect for further optimization.

B. MLIR-Based Compilation Pipeline

Arc-Lang leverages the Multi-Level Intermediate Representation (MLIR) framework [19] to represent and transform the program logic in a modular and extensible manner. The compilation pipeline consists of the following stages:

1. Dialect Construction: Arc-Lang defines a custom dialect named arc, consisting of types such as arc.stream, arc.struct, and arc.enum. These abstractions map directly to high-level Arc-Lang constructs.
2. Operator Modeling: Operations on these types—e.g., filtering, joining, mapping, aggregating streams—are encoded as dialect-specific MLIR ops with explicit semantics.
3. Transformation Passes: Several reusable passes are defined, including:
   - Canonicalization and CSE: Simplify IR via algebraic rewriting and eliminate redundant subexpressions.
   - Control Flow Structuring: Convert between basic blocks and structured control constructs (MLIR scf dialect) to suit different optimization and code generation targets.
   - Fusion and Inlining: Fuse stream operations to minimize memory usage and reduce runtime latency.
4. Lowering to Rust Dialect: Once optimized, the IR is converted to the rust MLIR dialect, which represents Rust constructs as first-class types and operations. Types are preserved as strings for use during code emission.

C. Rust Code Emission

The Rust dialect serves as a bridge between the MLIR world and native Rust code. Once the IR is lowered to this dialect, it is converted to human-readable and compilable Rust code.

Key characteristics of the Rust generation phase include:

- Name Mangling: User-defined aggregate types are assigned unique Rust-compliant names to prevent collisions.
- Ownership Model Integration: Arc-Lang does not expose Rust's ownership and borrowing system directly; instead, runtime macros handle memory management behind the scenes.
- Runtime Macros: Macros abstract constructs like concurrency, memory tracking, and stream manipulation. This prevents verbose boilerplate in generated code.
- Rustfmt Support: Formatting is delegated to rustfmt, ensuring readable and idiomatic code output.

D. Arc-Runtime System

To support distributed execution, Arc-Lang relies on a dedicated runtime library known as arc-runtime. This library implements core services such as:

- Stream Processing Engine: Manages time-ordered event streams and supports windowing, backpressure, and

event replay.
- Task Scheduler: Coordinates distributed task allocation using cooperative or centralized strate- gies.
- Fault Recovery: Implements snapshot-based checkpointing and recovery to handle node failures.
- Communication Layer: Uses asynchronous message passing over TCP or gRPC for inter-node coordination.

All concurrency is implemented using the tokio async runtime, ensuring scalable non-blocking IO. The runtime is modular and can be configured to deploy on standalone clusters, Kubernetes, or cloud-native serverless functions.

E. Testing and Verification

Arc-Lang uses MLIR's built-in tooling infrastructure to test IR transformations and compiler invariants:

- lit Tests: Regression and unit tests are defined using lit, which checks the correctness of output IR and verifies error diagnostics.
- Error Location Mapping: The compiler tracks source-to-IR mappings to generate meaningful error messages during compilation.
- Rust Unit Tests: Generated Rust code is embedded with #

$$cfg(test)$$

blocks to validate runtime behavior.

F. Toolchain and CLI

Arc-Lang provides a command-line tool named arc-mlir, built using the MLIR command-line interface infrastructure. It supports the following operations:

- lower-to=rust: Converts optimized IR into Rust code.
- run-pass=pass1,pass2: Allows users to select specific optimization passes.
- emit-ir: Outputs MLIR in human-readable textual format for inspection.
- verify: Validates structural invariants and typing rules within IR. This modular tool design allows for integration with CI pipelines and IDEs.

V. RESULTS

This section presents the experimental evaluation of Arc-Lang. The primary goals are to assess its effectiveness in (1) compiling high-level DSL programs to efficient Rust code, (2) enabling scalable distributed execution, and (3) achieving performance parity or improvement compared to established DSLs and frameworks. We conducted experiments across micro-benchmarks, code generation metrics, and real-world analytics pipelines.

A. Experimental Setup

All experiments were conducted on a 5-node cluster, each node equipped with an 8-core Intel Xeon CPU, 32 GB RAM, and running Ubuntu 22.04. Nodes were connected via a 1 Gbps Ethernet switch. Arc-Lang programs were compiled to Rust and then executed using the Arc-Runtime library powered by tokio. Comparative systems included Apache Beam (Java), Spark SQL (Scala), and TVM (Python).

B. Benchmarked Workloads

We evaluated three representative analytics tasks:

- Streaming Word Count: Ingesting a live stream of words and maintaining windowed frequency counts.
- Graph PageRank: Iterative rank propagation over a web-scale directed graph (100M edges).
- Tensor Aggregation: Summation and reduction over 3D tensor batches using user-defined aggregation functions.

C. Performance Metrics

We measured:

- Compilation Time (CT): Time to compile Arc-Lang to Rust (excluding 'rustc' compile).
- Execution Time (ET): End-to-end job runtime under steady-state throughput.
- Lines of Code (LOC): Output Rust code complexity (indicative of boilerplate).
- Throughput (TP): Events processed per second in streaming workload.
- Scalability (SC): Execution time relative to the number of nodes (2–5 nodes).

D. Comparison Table

Table 2 compares Arc-Lang with Beam, Spark SQL, and TVM across the three workloads.

Table 2: Performance Comparison of Arc-Lang and Other DSLs

System	CT (s)	ET (s)	TP (k/s)	LOC	SC (%)
Streaming Word Count					
Arc-Lang	2.1	9.4	**650**	**140**	**92**
Beam	N/A	11.8	420	340	81
SparkSQL	N/A	10.2	500	280	85
Graph PageRank					
Arc-Lang	3.6	22.5	-	**210**	**88**
SparkSQL	N/A	29.4	-	370	78
TVM	4.0	24.8	-	280	80
Tensor Aggregation					
Arc-Lang	2.3	8.1	-	**160**	**94**
TVM	2.9	9.7	-	240	88
SparkSQL	N/A	12.0	-	310	82

E. Findings

a) Compilation Efficiency

Arc-Lang compiles high-level programs to Rust in under 4 seconds for all workloads, benefiting from MLIR's modular transformation passes and reduced backend overhead.

b) Runtime Performance

Arc-Lang consistently outperforms Beam and Spark SQL in execution latency. This improvement stems from operator fusion, stream-aware scheduling, and efficient Rust- based executables.

c) Code Minimalism

The generated Rust code is significantly smaller in terms of LOC due to the use of runtime macros and structured code generation. Arc-Lang avoids boilerplate commonly found in Java- or Scala-based DSLs.

d) Scalability

Arc-Lang scales efficiently across distributed nodes, with minimal overhead due to its event-driven architecture and snapshot-based recovery model. The PageRank task, in particular, achieved an 88% speed-up from 2 to 5 nodes.

e) Expressiveness

Arc-Lang supports all data types uniformly—streams, tensors, and graphs—without requiring users to switch between DSLs. This simplifies cross-domain analytics.

VI. DISCUSSION

The experimental evaluation described in Section 5 demonstrates that Arc-Lang is a significant im- provement over existing domain-specific languages, including those of other popular games, in terms of performance, expressiveness, and cross-domain compatibility. This section analyzes the architectural choices found in Arc-Lang, the compromises made while designing and implementing the language, its current limitations, and the broader implications of its development.

In addition to this crucial feature, Arc-Lang enables the seamless combination of different data abstractions – streams, graphs, tensors, and tabular data – in one programming framework. By unifying these two entities, building hybrid analytical workflows which otherwise require separate frameworks (e.g., Apache Beam for stream processing, and Spark for batch analytics) becomes less complex. Arc-Lang improves developer productivity and reduces integration complexity by eliminating the need to switch contexts with heterogeneous systems. Nevertheless, such a generalization can induce more problems at the compile time to maintain type-safety and semantic compatibility of different data representations. It requires more advanced type system and compiler infrastructure than domain-specific languages oriented to narrow problems.

The choice of MLIR for the base compiler framework is essential for modularity and extensibility. A custom Arc dialect can be introduced which allows Arc-Lang to carry out domain-specific trans- formations along with existing optimization passes like constant folding, canonicalization, common subexpression elimination, etc. Even with these advantages, MLIR has a steep learning curve. Creat- ing specialized dialects and transformation passes entails knowledge of compiler internals, SSA, and how to compose dialects. Moreover, the intermediate representation of MLIR while best suited for debugging and optimization, is a little verbose and appears less appealing to the developer. :contentRef- erence[oaicite:0]index=0.

Arc-Lang compiles to Rust, which entails providing strong guarantees about type safety, memory management, and concurrency. (19 words) The generated code is efficient and compact, particularly in streaming workloads. Rust also offers suitability for deployment in edge and serverless environments. But there are challenges in using an ownership and

borrowing-matter language with Arc-Lang for high- level access. The Arc-Runtime layer was used to encapsulate the memory handling using macros and asynchronous constructs. This abstraction certainly saves users from dealing with low-level complexity, but it might expose some limitations when finer memory control is necessary.

Arc-Lang shows great performance advantage in streaming and tensor-based workloads according to analyses. Thanks to the use of operator fusion, zero-copy data movement, and asynchronous execution, these speedups were possible. The system is executed by events driven by pull-based. It suits modern-day distributed data processing system well. In the domain of graph analytics, though, whereas Arc-Lang shows that it can scale in a competitive way, it is not yet better than and specialized graph processors such as GraphX. The lack of optimizations specific to a domain, such as info locality-aware partitioning, incremental update propagation, etc. Although this is the case, Arc-Lang leverages lower coordination and serialization overhead.

The current implementation still has a few limitations. Because it is merely a research prototype, Arc-Lang does not have proper IDE support. So, there is no syntax highlighting, no static analysis and no interactive debugging tools. Furthermore, it can be difficult to debug the Rust code generated by an expression, as there is no strong correspondence between DSL constructs and generated code. Although the graph processing works, there's a need for better performance through message batching, adaptive partitioning, and other specialized joins. Moreover, the macro-based memory management approach in Arc-Runtime, which is convenient, may incur a small runtime overhead relative to manual optimizations.

Arc-Lang shows that it is possible for a high-level language and a type-safe DSL to implement multiple analyses through a unique framework. Looking at it from a systems perspective shows that using MLIR as an intermediate layer allows to build extensible and reusable language infrastructures. To the developers, Arc-Lang helps save time on learning multiple tools, libraries, and frameworks, and also in development and deployment cycles.

In short, Arc-Lang is a new programming model designed to make it easier to express and run data-intensive applications. The design give us insight for future research on unified analytics systems, providing evidence that abstraction, safety and performance can be combined in one language system, and it is possible.

VII. CONCLUSION

In this paper, we presented Arc-Lang, a high-level, type-safe domain-specific language designed to simplify and unify data analytics across diverse data modalities, including streams, graphs, tensors, and tabular data. Through a carefully layered architecture—comprising a typed DSL frontend, an extensible MLIR-based intermediate representation, and a performant Rust code generator—we demonstrated that it is possible to bridge the gap between expressiveness and execution efficiency.

Arc-Lang addresses several persistent challenges in the data analytics landscape. It eliminates the need for developers to manually orchestrate multiple DSLs by offering a unified programming model. It introduces strong typing across high-level data constructs, enabling early error detection and semantic correctness. Its integration with MLIR facilitates modular compilation and optimization, while its Rust backend ensures type safety, memory efficiency, and portability across platforms.

Empirical evaluation confirms that Arc-Lang delivers competitive performance across representative analytics workloads. The system achieves significant gains in compilation speed, runtime throughput, and scalability—especially in stream and tensor-based computations—when compared with established platforms such as Apache Beam, Spark SQL, and TVM. Moreover, the code generated by Arc-Lang is lean and readable, thanks to Rust's macro system and the compiler's structural code emission strategies. Importantly, the methodology behind Arc-Lang offers a reusable blueprint for developing next- generation DSLs. The combination of MLIR for intermediate abstraction and Rust for execution reveals a powerful synergy between language expressiveness and runtime guarantees. Arc-Lang's success reaffirms the growing relevance of IR-based compiler infrastructure in unifying programming models
for heterogeneous and distributed computing.

While several areas remain for enhancement—such as richer IDE tooling, advanced graph analytics optimizations, and source-level debugging—the current implementation establishes a strong foundation. It sets the stage for future exploration into cross-domain language design, just-in-time optimization, and lightweight deployment strategies.

Arc-Lang stands as a testament to the viability and benefits of designing data-centric languages that are both high-level and performance-conscious. By offering clarity, modularity, and safety without sacrificing speed, Arc-Lang advances the state-of-the-art in DSL design for scalable data analytics.

VIII. FUTURE WORK

Arc-Lang is a good start on a single unified extensible domain-specific language DSL for data analytics; however, there several interesting ways to boost its features. Future research will focus on optimization, execution adaptability, and developer tooling.

One suggestion for improvement is using modern graph optimization methods. While Arc-Lang includes a type system and streaming abstractions that make it easy to compute over graphs, it currently has no graph processing framework optimizations. Future efforts will implement mechanisms for incremental computation such as deltas which allow changing the state without having to recompute the whole graph. We will also effectively utilize vertex partitioning with partition-aware scheduling strategies to reduce communication costs. Combining hybrid execution methods will make it easier to analyze graphs efficiently at a larger scale. There will be two models: synchronous or asynchronous. The improvements should put the performance of Arc-Lang on par with that of Pregel, GraphX and PowerGraph.

A crucial next step is to add just-in-time compilation and adaptive optimization. Arc-Lang has been written in Rust, which offers strong static optimization guarantees. However, runtime flexibility is sacrificed as a result. Using MLIR's run-time capabilities for JIT-based compilation that specializes execution according to workload properties can be helpful. Based on execution feedback, we would be able to manually reorder, merge, and tune an operator. A good adaptive optimization mechanism can be useful in a dynamic environment such as a streaming data application and a real-time analytics system. Another vital area of future work is improving the developer experience, especially through strong IDE integration and tooling support. By integrating functionalities such as syntax highlighting, intelligent code completion, debugging tools, and performance visualization, it would be more user-friendly and efficient for developers employing arc-lang. This could drop the learning curve and increase adoption with improved debugging tools and well-integrated programming environments.

The future updates will ensure Arc-Lang gets faster, flexible and efficient to cater to the futuristic needs of applications that would rely on data.

IX. REFERENCES

[1] A. S. Foundation, Apache beam: An advanced unified programming model, Apache Beam Documentation (2017).

[2] K. Team, Keras: The python deep learning api, https://keras.io (2021).

[3] N. Francis, A. Green, P. Guagliardo, L. Libkin, et al., Cypher: An evolving query language for property graphs, Proceedings of the 2018 International Conference on Management of Data (2018) 1433–1445.

[4] A. Thusoo, J. Sarma, N. Jain, et al., Hive: A warehousing solution over a map-reduce framework, Proceedings of the VLDB Endowment 2 (2009) 1626–1629.

[5] M. Armbrust, et al., Spark sql: Relational data processing in spark, Proceedings of the 2015 ACM SIGMOD (2015) 1383–1394.

[6] P. Carbone, et al., Apache flink: Stream and batch processing in a single engine, IEEE Data Engineering Bulletin 38 (2015).

[7] A. Toshniwal, et al., Storm@twitter, in: ACM SIGMOD, 2014, pp. 147–156.

[8] J. Kreps, et al., Kafka: a distributed messaging system for log processing, NetDB (2011).

[9] T. Akidau, et al., The dataflow model: A practical approach to balancing correctness, latency, and cost, Queue 13 (2015).

[10] M. Abadi, et al., Tensorflow: A system for large-scale machine learning, OSDI (2016).

[11] A. Paszke, et al., Pytorch: An imperative style, high-performance deep learning library, NeurIPS (2019).

[12] T. Chen, et al., Tvm: An automated end-to-end optimizing compiler for deep learning, OSDI (2018).

[13] C. Leary, T. Wang, Xla: Optimizing compiler for machine learning, TensorFlow Dev Summit (2017).

[14] M. A. Rodriguez, The gremlin graph traversal machine and language, Proceedings of the DBPL Workshop (2015).

[15] T. Team, Gsql: A graph query language for tigergraph, https://www.tigergraph.com (2020).

[16] G. Malewicz, et al., Pregel: A system for large-scale graph processing, SIGMOD (2010).

[17] R. Xin, et al., Graphx: A resilient distributed graph system on spark, First International Workshop on Graph Data Management (2013).

[18] J. Gonzalez, et al., Powergraph: Distributed graph-parallel computation on natural graphs, OSDI (2012).

[19] C. Lattner, et al., Mlir: A compiler infrastructure for the end of moore's law, arXiv preprint arXiv:2002.11054 (2021).

[20] S. Palkar, et al., Weld: Rethinking the interface between data-intensive libraries, CIDR (2017).

[21] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis, CGO (2004).

[22] J. Ragan-Kelley, et al., Halide: A language and compiler for optimizing image processing pipelines, PLDI (2013).

[23] Z. DeVito, et al., Terra: A multi-stage programming language for high-performance computing, PLDI (2013).
[24] Y. Yu, et al., Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language, OSDI (2008).
[25] D. G. Murray, et al., Naiad: A timely dataflow system, SOSP (2013).
[26] P. Moritz, et al., Ray: A distributed framework for emerging ai applications, OSDI (2018).
[27] M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, Proceedings of the Python in Science Conference (2015).
[28] M. Contributors, Mars: A tensor-based unified framework for large-scale data computation, Open Source Project (2020).