Towards Expressive, Performant, and Correct Database Systems

Anna Herlihy
Supervised by Anastasia Ailamaki and Martin Odersky
EPFL
Lausanne, Switzerland
anna.herlihy@epfl.ch

ABSTRACT

Modern applications increasingly rely on expressive workloads that fall outside the relational core. These workloads expose a visibility problem: the information needed to verify, optimize, and execute them is often present somewhere in the system, but unavailable at the layer that needs it. I argue that database systems can support expressive workloads without sacrificing performance or correctness by recovering and propagating this information across system boundaries. My thesis pursues this goal by developing approaches that integrate classic data management techniques, programming language design, and formal reasoning across system layers.

VLDB Workshop Reference Format:
Anna Herlihy. Towards Expressive, Performant, and Correct Database Systems. VLDB 2026 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

Software systems are undergoing a fundamental shift. Modern applications are no longer written as monolithic programs over a single abstraction. Instead, they are assembled from off-the-shelf libraries, external tools, and specialized runtimes. These applications increasingly rely on computations that fall outside classical relational database abstractions: for example, they may require fixed-point computations, stateful operators, or complex statistical analyses written in general-purpose programming languages. The consequence for database systems is a class of workloads that cannot be naturally expressed or efficiently optimized using prevailing programming and execution models.

These workloads force a tradeoff between expressivity, performance, and correctness because the information needed to verify and optimize them is fragmented across abstraction boundaries between programming languages, query compilers, and execution engines. My thesis is that this tradeoff is not fundamental: by recovering and propagating information across abstraction boundaries, database systems can support expressive workloads while improving performance and guaranteeing correctness.

2 THE VISIBILITY PROBLEM

The Database Contract. Database systems offer users a powerful contract: express their computation using the classic relational database abstractions, namely relational algebra with aggregations and grouping, and the system will deliver correct results with good performance. Yet this contract breaks down as modern applications increasingly need *expressive workloads*, i.e., computations that do not fit neatly into the relational core. These computations may be technically expressible using SQL, but impractical. For example, it is far easier to reach for an existing library written in Python than to rewrite complex statistical analyses such as image recognition, audio transcription, or sentiment analysis in SQL. Alternatively, the computation may be inexpressible using relational algebra with aggregation and grouping alone. Examples include reachability and shortest-path analysis, hierarchical data such as bills-of-materials, and certain graph, security, network, and program analyses. In both cases, applications need more ergonomic and expressive capabilities than the relational core provides.

Paths to expressive workloads. Users have two options when their workloads fall outside the relational core. The first is to leave the database model and reach for general-purpose programming languages, composing pipelines from off-the-shelf utilities or user-defined functions (UDFs). This path is easy-to-use, modular, and convenient, but it leads to a *visibility problem*: optimizers need access to the structure of a program to apply optimizations, and external code does not expose that structure to the database. UDFs in particular incur order-of-magnitude slowdowns precisely because they are opaque to the optimizer, with the performance recoverable by translating them back into relational form [6].

The second path is to wrestle the computation into a form that the query optimizer has visibility into. The key database-native mechanism for increasing the expressivity of workloads is recursion [5]. SQL added the `WITH RECURSIVE` keyword in SQL:1999, and the same idea is central in Datalog, a logic-based query language. Queries with recursion have been used to express reachability, hierarchical data, program analysis [7], as well as machine learning pipelines. With a recursive operator, SQL becomes Turing-complete, capable of expressing any computable query. Recursive SQL has even been used to push the boundaries of what computations can be done within a database, for example implementing a GPT model or ray-tracer in SQL. Usability and performance aside, any computation a developer cares to encode can be expressed using recursion.

Yet encoding expressive workloads using recursion does not eliminate the visibility problem. Query optimizers rely on heuristics about the data stored in the database to inform the cost models that drive query optimization decisions. Recursive queries complicate this model because they are evaluated iteratively: each step applies the query to the output of the previous step until no new information is produced. As a result, the input data to each iteration is produced at runtime and may evolve unpredictably across iterations. Statistics gathered about the data stored in the database may only be relevant in the first iteration, while recursive queries

Proceedings of the VLDB Endowment. ISSN 2150-8097.

can have hundreds of iterations, and optimization decisions made in later iterations may no longer produce high-quality query plans. So while expressing computation using recursion means that the program structure is visible to the optimizer, the visibility problem has just moved to different, equally important information required for query optimization: the data.

Modern software trends. The visibility problem is becoming more acute because modern software development increasingly separates where code is written from where and how it runs. In cloud environments, dynamic resources and heterogeneous hardware make key optimization decisions depend on runtime conditions that are not visible at design time. Consider a data pipeline, executed in the cloud, that calls a UDF. The UDF is written in a general-purpose programming language that calls a domain-specific library, for example for a particular statistical analysis. The pipeline author may not have the domain expertise to reimplement or optimize the analysis, and so uses an off-the-shelf library; the library author does not know how their code will be composed into a pipeline or what hardware it will run on; and the database system, which has access to the hardware and runtime environment, sees only an opaque function call. As a result, no single entity has enough visibility to optimize the entire pipeline effectively.

This challenge has been gaining momentum for years, but the introduction of large language models (LLMs) as sources of code, from text-to-SQL to fully generated applications and LLM-guided systems, marks a critical inflection point. LLMs lower the barrier to producing complex programs, but the resulting code may be unreliable: it can be syntactically plausible while still encoding incorrect assumptions about semantics, performance, or the execution environment. Existing systems provide limited support for enforcing performance or correctness contracts that shape code generation and execution. If programs cannot be assumed efficient, correct, or even written in a disciplined way, it is even more important for systems to gain visibility into code to recover optimization opportunities and enforce strong correctness and performance guarantees. These trends point to a deeper systems problem: the information needed to verify, optimize, and execute expressive workloads is spread across layers of abstraction.

Key information is hidden by abstraction boundaries. Modern data systems are organized around abstractions, illustrated in Figure 1. At the specification layer, abstractions such as languages, type systems, and static analyses describe what a computation should do and what properties can be derived offline before execution, including semantic and correctness properties. At the translation layer, system components such as query optimizers and compilers turn these specifications into executable strategies through logical rewrites, physical planning, and code generation. At the execution layer, runtimes execute those strategies on concrete data and hardware, where resource availability, intermediate results, runtime behavior, and bottlenecks become visible. The interfaces between these layers are the abstraction boundaries across which key information must travel, and the resulting fragmentation is the visibility problem: key information is not available at the layer of abstraction that needs it. Thus the unifying question of my research is: *when computation crosses an abstraction boundary, what opportunities are lost, and where can we recover them?*

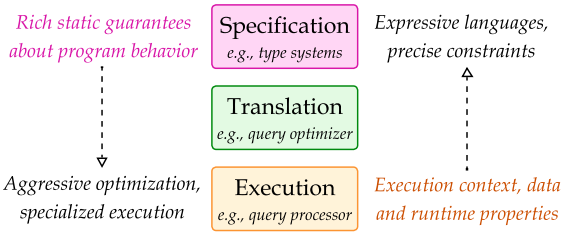

Figure 1: Key information across abstraction boundaries.

```
base="https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year";
for y in {2010..2020}; do
  curl -s "$base/$y.csv.gz" |
  gunzip | noaa-decoder | range-calc |
  sed -e "s/ *$//" | xargs -I{} grep {} ghcnd-stations.txt |
  sed "s/^/Station with the largest temperature range in $y:"
done
```

Figure 2: Example analytical pipeline with a mix of utilities.

3 EXPRESSIVITY WITHOUT REGRET

The consequence of the visibility problem is a tradeoff between expressivity, performance, and correctness. Systems may simply not be expressive enough, so users move the computation outside the database and leave performance on the table. Systems may be expressive enough but slow: a small, well-supported core surrounded by an "escape-hatch" of less-optimized features that users may slip into without realizing, leading to expressivity at the cost of unpredictable performance. Or systems may be expressive enough but unsafe, leading to runtime exceptions, nontermination, or undefined behavior that silently returns incorrect results. The goal of my research is *expressivity without regret*: database systems that are simultaneously expressive, performant, and correct. The mechanism is to identify where relevant missing information lives and move it to the layer that can use it most effectively.

3.1 Automatic Extension of DBMS Functionality

To address the first path, leaving the relational core to compose pipelines from external utilities, I proposed Generalized OLAP (GOLAP) [2]. GOLAP is a new DBMS paradigm that places automatic extensibility of functionality as a first-class design goal. Modern applications often need domain-specific functionality unavailable as a native database operator; reusing existing libraries provides it with minimal effort, but moves the computation outside the query optimizer's visibility.

Figure 2 shows an example data analysis pipeline on weather data. The pipeline is easy to assemble from existing components, but suffers from unnecessary serialization and hardware underutilization. These inefficiencies are not caused by the internal implementation of any single utility, they arise because no component has visibility across the full pipeline. Unlike prior work that optimizes a utility's internal logic (translating a UDF back to relational form [6] or lifting it into an optimizable IR), GOLAP targets the inefficiencies *between* utilities, which no single utility's optimizer can see.

The GOLAP prototype ingests external utilities and optimizes their execution using static analysis and runtime instrumentation to detect inefficiencies near utility boundaries, rewriting utilities

```
WITH RECURSIVE WaitFor AS
  (SELECT part, days
  FROM BasicParts
    UNION
  SELECT sp.part, MAX(wf.days)
  FROM SubParts sp, WaitFor wf
  WHERE sp.sub = wf.part
  GROUP BY sp.part);
SELECT * FROM WaitFor
```

(a) Non-monotonic query: Bill-of-materials

```
WITH RECURSIVE Path AS
  (SELECT * FROM Edges
    UNION
(WITH Path as -- Postgres
    (SELECT * FROM Path)
  SELECT p1.x, p2.y
  FROM Path p1, Path p2
  WHERE p1.y = p2.x));
SELECT * FROM Path
```

(b) Non-linear query: Transitive closure

```
WITH RECURSIVE Gens AS
  (SELECT p.ch as nm, 1 as g
  FROM Parents p WHERE p.par='A'
    UNION ALL
SELECT p.ch as nm, g.g+1 as g
  FROM Parents as p, Gens as g
  WHERE p.par = g.nm);
SELECT * FROM Gens
WHERE Gens.g = 2
```

(c) Bag-semantic query: Same-Generation

	WITH REC.	Range-Rest.	Agg.	Mut.	Non-linear	Set	Const-Free
MySQL	✓	✓	✗	✗	•	✓	✓
OracleDB	✓	✓	✗	✗	✗	✗	✓
PostgreSQL	✓	✓	✗	✗	✗	✓	✓
SQL Server	✓	✓	✗	✗	•	✗	✓
SQLite	✓	✓	✗	✗	•	✓	✓
MariaDB	✓	✓	✗	✓	✓	✓	✓
DuckDB	✓	✓	✓	• (<v1.2)	•	✓	✓

(d) Support for recursion on modern RDBMS. ✗ Prevented, • OK, ✓ Supported.

Figure 3: Examples of dangerous recursive queries and their support across modern RDBMS.

or applying hardware-aware optimizations to improve execution. It shows up to 22x speedup on the NOAA pipeline by bypassing serialization and specializing execution.

This work derives key information from the specification layer and makes it available to the execution layer to enable expressive, modular, and efficient pipelines. However, GOLAP recovers structure after a pipeline has already been written. The next work asks a complementary question: can we design expressive database abstractions so that information is preserved across layers?

3.2 Type-Safe Recursive Query

To address the second path to expressive workloads, i.e., expressing computation inside the database using recursion, I proposed TyQL, a language-integrated query system for safe recursive SQL [4]. Language-integrated query, popularized by LINQ, aims to combine the ergonomics of a general-purpose programming language with the performance of a RDBMS. Queries are expressed natively in the host language so the compiler can statically catch errors while compiling to SQL for backend execution. Yet no existing library supports recursion, leaving expressive workloads without correctness or safety guarantees. Rather than introduce a new query language, which would ask developers to leave SQL and its ecosystem behind, my approach composes with existing SQL systems and with widely used programming languages such as Scala, bringing compile-time safety to recursive queries through the host language's type system so they run on unmodified backends.

I first identified how recursive queries risk correctness beyond ordinary SQL errors: certain runtime database exceptions, incorrect results, and nontermination. I then identified the six mathematical properties of recursive computation that are responsible for these behaviors: range-restriction, monotonicity, mutual-recursion, linearity, set-semantics, and constructor-freedom. Figures 3a-3c show queries that can lead to unwanted behavior. The bill-of-materials query is non-monotonic as it contains aggregation inside recursion, and may trigger a runtime database exception. The transitive-closure query is non-linear as it references the recursive relation more than once, and may silently return incorrect results. The same-generation query uses bag semantics and a constructor (+), which can cause nontermination.

Figure 3d shows why these properties cannot be checked by targeting a single uniform model of recursive SQL. The table compares support for recursive-query features across modern RDBMS: ✗ indicates that the feature is unsupported and will throw a relevant error message, while • indicates *syntactic* support, meaning the query will not throw an error even though the system may not have the internal implementation needed to execute it correctly. The technical landscape is wide and ever-growing, and no two modern databases support the same set of features. The lack of alignment among commercial RDBMS renders a single, unified formal semantics untenable as a foundation for correctness checking of recursive queries; practical systems must instead support a flexible and composable framework that adapts to the specific capabilities of each backend.

To solve this problem, I developed a calculus, λ_{RQL}, that enforces correctness across the programming language and database boundary through composable mathematical properties derived at compile time. I proved that queries satisfying these properties are guaranteed to terminate with the correct result in a finite number of steps. I implemented this in TyQL, a practical library that enforces correctness through the type system. I also developed a recursive-query benchmark and RDBMS survey to evaluate query coverage and backend behavior, showing that TyQL prevents unsafe queries without adding overhead compared with raw SQL strings.

TyQL therefore targets expressivity and correctness: users can express recursive computations inside the database, while the system enforces the properties needed for safe execution. However, it does not target performance, which motivates the next work.

3.3 Adaptive Recursive Query Optimization

Traditional relational query optimization techniques do not scale well to recursive query processing because recursive evaluation is iterative and the optimal query plan may not remain optimal after the first iteration. To address this problem, I introduced Adaptive Metaprogramming, a technique that shifts recursive query optimization and code generation from compile time to runtime using principled metaprogramming [3]. The key idea is to make and continuously revisit optimization decisions as information becomes available. Figure 4 illustrates this staged view of recursive query execution: some information is available when the query is compiled, some appears when the input data is known, and other information, such as intermediate data and slow operations, becomes available only after execution begins.

I implemented this approach in Carac, a recursive query engine that supports runtime optimization and repeated re-optimization.

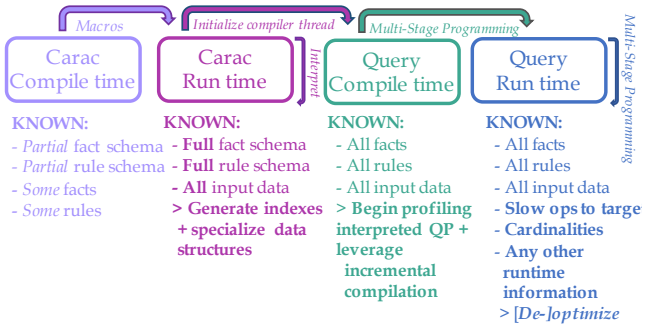

Figure 4: Stages information becomes available in Carac

Figure 5: Bidirectional flow of information in CaQL

Unlike adaptive query-processing systems that switch among precompiled plans, Carac uses metaprogramming and runtime information to regenerate plans and reoptimize recursive queries across iterations. Carac illustrates the impact of adaptivity on recursion by targeting join ordering for conjunctive subqueries. In the evaluation, Adaptive Metaprogramming improved unoptimized recursive query execution by three orders of magnitude and outperformed manually hand-optimized queries in several cases.

Carac targets expressivity and performance. Together, TyQL and Carac recover information in opposite directions. TyQL flows properties from the specification to the execution layer, while Carac flows runtime observations up to the translation layer. The natural next question is whether connecting these two directions can improve expressivity, performance, and correctness at the same time.

3.4 Bidirectional Information Flow

To target expressivity, performance, and correctness simultaneously, I proposed CaQL, a recursive query framework that combines TyQL and Carac through bidirectional information flow between layers of abstraction [1]. Figure 5 illustrates this: query properties flow from the type system to the execution engine, enabling more aggressive optimization when the type system has proved that an execution strategy is safe. In the other direction, data and execution-environment properties flow from the execution engine back to the type system, enabling the system to accept queries that would otherwise be rejected. Where information flowing toward the execution layer unlocks performance, information flowing back to the specification layer unlocks expressivity: relaxing a constraint is equivalent to adding new functionality to the system.

For example, linearity can be treated both as a safety property and an optimization opportunity. If the type system proves that a query is linear, the execution engine can safely use a more efficient evaluation strategy. If the query is non-linear, the execution layer can signal to the type system that it can use a runtime that preserves the intended recursive semantics. The type system can then safely relax its linearity check and accept queries it would otherwise reject.

CaQL therefore targets expressivity, performance, and correctness simultaneously: recursive queries are guaranteed to be safe, while the system selects execution strategies that improve performance. In evaluation, CaQL provides the safety and convenience of language-integrated query with performance competitive with or superior to a state-of-the-art Datalog engine.

3.5 Next Steps

In an increasingly AI-driven technology landscape, system components may be unreliable, and may encode incorrect assumptions about semantics, performance, or the execution environment. Future systems may run without any human checks at all, so visibility across components becomes more important than ever. When a system accepts a program, that acceptance should carry explicit, checkable guarantees on correctness and resource use.

In the final year of my PhD I want to extend the ability of database engines to derive more advanced properties from the programs they execute. Unpredictable performance often stems from assumptions about resource use that do not hold at runtime, so the goal is to encode resource-usage constraints in the programming language through substructural type systems. Existing substructural type systems, such as linear types, are rigid: they apply a single uniform discipline across an entire program, making them poorly suited for the varied resource requirements of real data systems. I am developing a more flexible and modular approach that lets different resource constraints coexist within the same program, starting with Linear Datalog as a case study.

Making these resource contracts as actionable as compilation errors provides a natural substrate for agentic code generation, allowing agents to use these contracts as a feedback signal. Over the longer term, this raises a deeper design question for the era of machine-generated code: should programming languages and data systems continue to be designed around assumptions about human authorship, or should they be rethought to explicitly support machine-generated code as a first-class input?

REFERENCES

[1] Anna Herlihy, Anastasia Ailamaki, and Martin Odersky. 2025. Static Typing Meets Adaptive Optimization: A Unified Approach to Recursive Queries. In *Proceedings of the 19th International Symposium on Database Programming Languages, DBPL 2025, Berlin, Germany, June 22-27, 2025*. ACM, 2:1–2:6. https://doi.org/10.1145/3735106.3736533

[2] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p81-herlihy.pdf

[3] Anna Herlihy, Guillaume Martres, Anastasia Ailamaki, and Martin Odersky. 2024. Adaptive Recursive Query Optimization. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 368–381. https://doi.org/10.1109/ICDE60146.2024.00035

[4] Anna Herlihy, Amir Shaikhha, Anastasia Ailamaki, and Martin Odersky. 2026. Language-Integrated Recursive Queries. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2026) (Leibniz International Proceedings in Informatics (LIPIcs))*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[5] Leonid Libkin. 2003. Expressive power of SQL. *Theor. Comput. Sci.* 296, 3 (March 2003), 379–404. https://doi.org/10.1016/S0304-3975(02)00736-3

[6] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2018. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (oct 2018), 432–444. https://doi.org/10.1145/3164135.3164140

[7] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226