TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science / Institute of Computer Engineering, Chair for Compiler Construction

An Extended Evaluator and Backend for Querying Bi-temporal Graph Databases

Bilal Mahmoud

bilal.mahmoud@mailbox.tu-dresden.de
Born on: 23.11.1999 in Dresden
Matriculation number: 4843857

Diplomarbeit

Supervisors
Felix Suchert

Supervising Professor
Prof. Dr.-Ing. Jeronimo Castrillon

Referee
Dr.-Ing. Alexander Krause

Submitted on: 21.04.2026

Statement of authorship

I hereby certify that I have authored this document entitled *An Extended Evaluator and Backend for Querying Bi-temporal Graph Databases* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 21.04.2026

Bilal Mahmoud

Contents

List of Figures ii

List of Listings iv

List of Tables vi

List of Acronyms viii

1. Introduction 1

2. Background 4

3. Related Work 6
- 3.1. Graph Query Execution . . . 6
- 3.2. Query Compilation and Intermediate Representations . . . 8
- 3.3. Heterogeneous Work Placement . . . 10

4. System Overview 13
- 4.1. Mid-Level Intermediate Representation . . . 15
- 4.2. HIR Normalization . . . 15
- 4.3. Target Placement . . . 16
- 4.4. Code Generation and Execution . . . 17

5. HIR in Administrative Normal Form 18
- 5.1. ANF . . . 18
- 5.2. Normalization . . . 20
- 5.3. Effect Hoisting . . . 24
- 5.4. Thunking . . . 26
- 5.5. Reification . . . 27

6. Mid-Level IR 30
- 6.1. Instruction Set . . . 32
  - 6.1.1. Statement . . . 33
  - 6.1.2. Terminator . . . 36
  - 6.1.3. Effectful Terminator . . . 37
- 6.2. Dataflow Analysis . . . 38
- 6.3. Transformation Pipeline . . . 40
  - 6.3.1. Inlining . . . 41
  - 6.3.2. Canonicalization . . . 45

7. Target Placement 58

7.1. Cost Model ... 59
7.1.1. Edge-Based Attribution ... 60
7.1.2. Block-Based Attribution ... 61
7.1.3. Objective Formulation ... 62
7.1.4. Separable Approximation ... 66
7.2. Statement Placement ... 67
7.2.1. PostgreSQL ... 69
7.2.2. Embedding ... 71
7.2.3. Interpreter ... 72
7.3. Transfer Cost ... 72
7.3.1. Size Estimation ... 73
7.4. Block Cost Assembly ... 76
7.5. Solving ... 78
7.5.1. Domain Pruning ... 81
7.6. Block Fusion ... 82
7.7. Islands ... 83

8. Code Generation and Execution 86
8.1. Interpreter ... 87
8.2. PostgreSQL ... 88
8.3. Orchestrator ... 93

9. Evaluation 94
9.1. Methodology ... 94
9.1.1. Corpus ... 94
9.2. Correctness ... 95
9.3. Compilation and Execution Performance ... 96
9.4. Filter API Comparison ... 98
9.5. Placement Effectiveness ... 99
9.6. Interpreter Performance ... 100

10. Conclusion and Future Work 103
10.1. Future Work ... 105
10.1.1. Execution Infrastructure ... 105
10.1.2. Pipeline and Backend Extensions ... 106
10.1.3. Language and Type System ... 107
10.1.4. Optimization ... 108

Bibliography 110

A. OFFSET 0 Materialization Fence: Plan Comparison 117
A.1. Source Expression ... 117
A.2. Generated SQL ... 118
A.3. Parameters ... 120
A.4. Query Plans ... 120
A.4.1. With `OFFSET 0` ... 120
A.4.2. Without `OFFSET 0` ... 120

B. Related Work Synthesis 122

List of Figures

Figure 1 Compiler Pipeline 14

Figure 2 ANF Syntax 19

Figure 3 Transformation Pipeline 40

Figure 4 Call graph and utility definitions 42

Figure 5 Body cost estimation 42

Figure 6 Loop breaker scoring 43

Figure 7 Callsite scoring 44

Figure 8 Inlining utility definitions 45

Figure 9 Canonicalization 46

Figure 10 Instruction simplification rewrite rules 53

Figure 11 Edge-based vertex path attribution strategies. 61

Figure 12 Block-based attribution with separable over-approximation. 62

Figure 13 Vertex path hierarchy (subset). Composites share their children’s origin. 78

Figure 14 Type-directed JSON serialization. 90

Figure 15 Compilation time breakdown for the all-entities query. 96

Figure 16 Thesis contribution to compilation time across query shapes. 97

Figure 17 Filter API versus compiled pipeline execution time, with and without type system overhead. 98

Figure 18 Execution time with solver placement versus forced interpreter execution. 99

Figure 19 Interpreter versus CPython 3.8 across four algorithms at increasing input sizes. 101

List of Listings

Listing 1 Effect hoisting: `%1` is promoted because it does not reference `v:0` 25

Listing 2 Thunking: top-level bindings wrapped, references become calls 27

Listing 3 Thin and fat call desugaring during reification 28

Listing 4 MIR program illustrating the instruction set 33

Listing 5 Load from a projection chain and from a constant 34

Listing 6 Binary addition and comparison 34

Listing 7 Bitwise not and arithmetic negation 35

Listing 8 Tuple, opaque wrapper, and closure construction 35

Listing 9 Loading and checking an input parameter 35

Listing 10 Direct and indirect function application 36

Listing 11 Unconditional jump with block arguments 36

Listing 12 Multi-way branch with an otherwise target 36

Listing 13 Return with a value 36

Listing 14 Unreachable block marker 36

Listing 15 Graph read with a filter predicate 37

Listing 16 Traversal-aware liveness on a diamond control flow graph (CFG) 39

Listing 17 Forwarding closure reduction: the wrapper is eliminated and the inner call is exposed 48

Listing 18 Data dependency graph for tuple projection and parameter propagation 50

Listing 19 Dead cycle: `%1` and `%3` depend on each other with no path to a root use 55

Listing 20 Generated query template for two compiled islands. 92

Listing 21 All Entities: Collect with a constant-true filter. 123

Listing 22 Filter by UUID: Single equality comparison against an input parameter. No branching. 123

Listing 23 Diamond CFG: Conditional filter with if-then-else. 124
Listing 24 Sequential Filters: Two chained filters with nested conditionals. 124
Listing 25 Stress Test: Three sequential filters. 124

List of Tables

Table 1 Short-circuit desugaring equivalence 23

Table 2 Logical and bitwise truth table 35

Table 3 Statement and terminator costs per execution target. 69

Table 4 Transition matrices by terminator kind. 73

Table 5 Fixed backend switch overhead per target pair. 73

Table 6 Test counts per compiler crate. 95

Table 7 Surveyed systems against the conjunction of properties this work addresses. 122

List of Acronyms

ANF. administrative normal form
AR. administrative reduction
AST. abstract syntax tree
BnB. branch-and-bound
CFG. control flow graph
CP. copy propagation
CPS. continuation-passing style
CSP. constraint satisfaction problem
CST. concrete syntax tree
CTE. common table expression
DAG. directed acyclic graph
DBE. dead block elimination
DLE. dead load elimination
DSE. dead store elimination
FS. forward substitution
GSA. gated single assignment
GVN. global value numbering
HIR. high-level intermediate representation
HIR/ANF. high-level intermediate representation in administrative normal form
ILP. integer linear programming
IR. intermediate representation
IS. instruction simplification
JIT. just-in-time compilation
LICM. loop invariant code motion
LP. linear programming
MIR. mid-level intermediate representation
MRF. markov random field
PBQP. Partitioned Boolean Quadratic Programming

QPBO. quadratic pseudo-Boolean optimization
SCC. strongly connected component
SCCP. sparse conditional constant propagation
SIL. Swift intermediate language
SRMP. sequential reweighted message passing
SSA. static single assignment
TRW-S. tree-reweighted sequential message passing
UB. undefined behavior

1. Introduction

The HashQL compiler, developed as part of the Großer Beleg, provides a complete frontend for querying HASH's bi-temporal knowledge graph: parsing, type inference, and type checking over a statically typed functional calculus with structural subtyping and first-class graph effects. The language is traversal-based rather than declarative: expressing bi-temporal constraints over a strongly typed graph through pattern matching alone is infeasible, so the language exposes traversal semantics with type-aware predicates and functional composition [11]. That compiler could execute only a subset of well-typed queries by translating them into HASH's internal filter API, which supports boolean connectives, comparison operators, and property path access over entity sets. Conditional expressions, aggregate construction, multi-hop traversals, and closures had no execution path. The evaluator was further limited in architecture: it targeted a single backend and provided no mechanism for distributing work across heterogeneous execution targets.

Because HashQL is a typed functional language that expresses graph operations as effects rather than relational algebra, traditional database placement techniques do not transfer directly: the unit of planning is not a relational operator but a typed instruction, backend capability depends on the type system, and transition costs are asymmetric. Standard compiler methodology provides the foundation instead: the program is treated as an ordinary typed program, and database systems become one compilation target among several, each accepting the subset of instructions it can represent.

HASH is a heterogeneous computational environment. Different backends store different facets of the graph: relational data in PostgreSQL, vector encodings in a dedicated embedding store. The existing system hardcodes each backend's integration: new data sources require dedicated wiring, cannot participate in a common execution framework, and cannot receive computation offloaded from the compiler. A universal interpreter solves the executability gap but not the efficiency gap: it forces every row through the runtime regardless of whether the backend could have evaluated the operation locally, and computation that PostgreSQL could perform inside the database engine is instead executed per-row by the interpreter. This work provides a compilation target model that addresses both: each backend declares which subset of the instruction set it can express, and the compiler colocates operations with the data they consume to minimize transfer cost.

The difficulty is that each target's computational model determines what it can express, and a single query body may span several. PostgreSQL accepts declarative SQL but rejects types it cannot serialize unambiguously; the interpreter handles the full instruction set but at higher per-instruction cost. An operation that is ineligible on one backend renders every consumer of its result ineligible as well, so eligibility propagates through data dependencies. The compiler must partition each body across backends, account for the cost of transferring data at every boundary, and coordinate execution at runtime through suspension and resumption.

The execution half of the compiler comprises four stages. The tree-structured high-level intermediate representation (HIR) hides evaluation order and per-statement data dependencies; placement at instruction granularity requires a representation where both are explicit. administrative normal form (ANF) normalization bridges the HIR to a statement-based representation by binding every subexpression to a fresh variable and fixing evaluation order (Chapter 5). A type-preserving static single assignment (SSA)-form mid-level intermediate representation (MIR) replaces the functional representation with a CFG of basic blocks suitable for transformation and analysis; effects are modeled as first-class terminators with suspension semantics, so the same representation serves optimization, placement, and runtime dispatch (Chapter 6). A solver over a separable cost model assigns each basic block to an execution backend under capability and transfer-cost constraints (Chapter 7). A runtime phase compiles each assigned island into backend-specific artifacts and dispatches them through an orchestrator; the interpreter handles residual computation via coroutine-style suspension for effectful terminators (Chapter 8).

The concrete contributions are:

1. A mid-level intermediate representation whose first-class effectful terminators model traversal operations as control-flow boundaries; type-preserving transformations over this representation; and a canonicalization pipeline that applies standard compiler optimizations to a query language.
2. A separable cost model that decomposes the placement problem into per-block computational cost and pairwise transfer cost, with a solver that assigns basic blocks to backends under capability constraints that propagate through data dependencies.
3. Two execution backends: a PostgreSQL code generator that compiles MIR basic blocks into parameterized SQL expressions, and a tree-walking interpreter with coroutine-style suspension for effectful terminators.
4. An orchestrator that dispatches islands to their assigned backends and mediates data transfer at island boundaries through demand-driven vertex hydration.

Chapter 2 and Chapter 3 supply the platform context and position this work against graph query execution engines, query compilation infrastructures, and heterogeneous placement systems. Chapter 4 presents the full pipeline. The four technical chapters follow the compilation order: Chapter 5 normalizes the HIR into administrative normal form; Chapter 6 defines the MIR instruction set and its transformation pipeline; Chapter 7 formalizes placement as a separable cost-model optimization; and Chapter 8 describes code generation and the orchestrator that coordinates heterogeneous

dispatch. Chapter 9 evaluates compilation overhead, placement effectiveness, and interpreter performance.

2. Background

HASH is an open-source, strongly typed, bi-temporal graph database. The Großer Beleg designed HashQL as a query language for the platform and built the compiler frontend [11]; this work continues from where that pipeline ends. Vertices in the graph belong to one of two layers: the ontology, which defines the vocabulary of types, and the knowledge graph, which stores instantiations of those types. The ontology is organized over three levels of abstraction: data types describe value domains and may be grouped, property types give semantic meaning to these domains or aggregate them into composite types by referencing other property types or forming recursive structures, and entity types specify entity schemas that are used to instantiate knowledge, declaring the links that are allowed, their cardinality, and the properties an entity may express. The ontology type system is a superset of JSON Schema; knowledge is therefore constrained to JSON-serializable values. Ontology items carry globally unique URL identifiers to permit collision-free composition across independently published vocabularies. Because both ontology and knowledge reside in the same graph, queries can traverse between data and its type definitions, which permits type-aware predicates and introspection at query time.

Links in the HASH graph are not a separate primitive, but are modelled as entities themselves. HASH separates the notion of a relationship from that of a connection: a link is an entity whose type extends a designated base link type, and it always carries exactly two outgoing edges, `left` and `right`, to its endpoints. Because links are entities, they participate in the graph like any other entity: they can carry properties, serve as endpoints of other links, and form higher-order relationships in which a link connects to another link. Subsequent chapters use “entity” without distinguishing data entities from link entities unless the distinction is relevant to a specific design decision.

The graph is bi-temporal along two axes, with the design admitting extension to additional axes. The first axis is transaction time, which records when a fact was stored. The second is decision time, which records when a decision about a specific entity was made; decision time is always $\leq$ transaction time. Decision time is distinct from the valid time axis found in other temporal database systems, which records when a fact holds true and may lie in either the future or the past. Each query is constrained on both axes: one is pinned to a specific point in time, and the other is variable, defining a time frame of validity. Geometrically, if transaction and decision time form the axes of a two-dimensional coordinate system, each query draws a line, and entities, represented as rectangles in that space, are selected when they intersect it. Temporality affects

queries in two respects: the language must support traversal across temporal axes, and temporal metadata constrains the result set, because which entities are reachable depends on the temporal window under which the query executes.

HashQL, developed as part of the Großer Beleg, is a query language over the HASH graph. It is a statically typed functional calculus that takes a traversal approach rather than a declarative one. The language core is syntax-independent; the design permits future surface syntaxes that target the same compiler pipeline. The Großer Beleg introduced a single frontend that continues to be used in this work: J-Expr, a JSON-with-comments format inspired by S-expressions. The underlying type system supports structural subtyping as the default and nominal opt-in. The type system reconciles the ontology types of the HASH graph, which it can represent directly, with common programming idioms such as dynamic containers. The language has no concept of statements: every expression denotes a value.

All interaction with the bi-temporal graph is isolated by the effect type `Graph<T>`. Computations that read from or write to the graph produce values of this type, which are first-class and composable but opaque: they can only be executed by the host runtime at the program boundary. This separation follows the standard monadic pattern for effect isolation. Graph queries compose through three kinds of operations: head functions initiate a query and produce the `Graph<T>` effect, body combinators transform a `Graph<T>` into a new `Graph<U>` without materializing intermediate values, and tail operations evaluate the graph computation and return a materialized result. The Großer Beleg defined `filter` as a body combinator and `collect` as a tail operation; this work continues to use them as the primary graph operations to determine the viability of the execution model.

The compiler pipeline preceding this work created three levels of abstraction that are successively lowered: a concrete syntax tree (CST) (J-Expr) is lowered into an abstract syntax tree (AST), which is subsequently lowered into the HIR. The AST resolves imports and formalizes special forms into dedicated nodes. The HIR preserves the tree-shaped structure of the source program while removing syntactic sugar, and serves as the basis for type inference and type checking. Both representations, and the type system that governs them, were developed as part of the Großer Beleg. The compiler at that stage could parse, type-check, and validate queries, but could not execute them beyond a limited filter-based evaluator that translated a small subset of queries into HASH's internal filter API.

The filter-based evaluator compiled `filter` bodies into HASH's internal filter representation, supporting boolean connectives, comparison operators, and property path access. Conditional expressions, aggregate literals, multi-hop traversals, and closures were not executable. The evaluator was further constrained in architecture: it targeted a single homogeneous backend and offered no mechanism for heterogeneous execution or compositional extension to new graph operations. A different representation and execution strategy were required to make the compiler viable beyond this narrow fragment.

3. Related Work

Compiling a typed, functional graph query language for heterogeneous execution requires infrastructure across three areas: a graph-aware execution model, an intermediate representation suitable for multi-target compilation, and a placement mechanism that assigns work to backends automatically. Each area individually has an established body of prior work, yet the requirements interact: the intermediate representation (IR) must preserve types so that placement can determine both eligibility and transfer costs, placement must respect backend capability constraints that differ across targets, and code generation must produce output in each target's native computational model. Because the architecture must generalize over arbitrary k backends, the placement and code generation infrastructure cannot assume a fixed set of targets. No surveyed system addresses all three requirements jointly.

3.1. Graph Query Execution

Graph query engines differ in compilation strategy, storage model, query language, and scale. Most share a common constraint: they assume that each query executes under a single computational model, a backend with its own execution semantics and expressible operation subset (Chapter 7 formalizes this notion as an execution target). Join-based engines such as Graphflow [58] and EmptyHeaded [2] rewrite pattern queries as worst-case-optimal multiway joins, achieving orders-of-magnitude speedup over traditional optimizers. Both are standalone engines that replace, rather than extend, existing graph databases. GRAPHITE [61] takes the opposite approach and builds graph traversal on top of a relational storage engine, pruning irrelevant edge fragments via a transition graph index, but supports only a single relational backend. Chimera [50] extends PostgreSQL with dual graph and relational storage and a unified Traversal-Join operator for SQL/PGQ queries. Despite the hybrid storage model, all operations execute within one extended RDBMS.

Distributed systems address scale but not the single-model constraint. Granite [68] targets temporal path queries via BSP wavefront expansion; it is the only system surveyed here that addresses temporality, though limited to uni-temporal (transaction-time) data. GraphDance [14] pipelines stateful traversal asynchronously, and GAIA [66] scales graph query execution across partitioned topologies. GAIA is one of the few systems that, like HashQL, adopts a traversal-based rather than a pattern-matching approach; this decision is motivated in the Großer Beleg [11]. None of these systems

can distribute operations across backends with different computational models within a single query.

Bingqing Lyu, et al. [53] introduce GOpt, a graph-native query optimization framework that decouples optimization from both query language and execution engine. Queries in Cypher or Gremlin are compiled into a unified Graph intermediate representation, optimized through a combination of heuristic rules, automatic type inference, and cost-based search, and then converted into backend-specific physical plans. Each backend registers its own operators and cost model: their work targets Neo4j for single-machine execution and GraphScope for distributed dataflow. GOpt validates that a unified graph IR with backend-specific code generation yields substantial performance gains. The framework is, however, a retargetable optimizer, not a heterogeneous execution planner: the converter produces a plan for one backend at a time. There are no mixed-backend plans within a single query, no transfer operators between backends, and no solver over backend assignments.

The G-SPARQL engine [73] and its distributed extension DG-SPARQL [8] are the clearest prior example of within-query heterogeneous execution for graph queries. G-SPARQL classifies its algebraic operators into two families: retrieval-based operators (attribute lookups, structural predicates, edge joins), which have direct relational equivalents and are compiled to SQL for execution inside an RDBMS, and traversal-based operators (reachability, shortest path, path filtering), which require recursive graph exploration and are executed by in-memory algorithms over a pointer-based topology representation. The split is evaluated at the algebraic plan level: a bottom-up traversal groups connected retrieval operators into SQL sub-plans, stopping at each traversal operator boundary. DG-SPARQL extends this to distributed settings with BSP-based traversal across partitioned graph topologies, while replicating the relational store on each node and using selectivity-based routing to parallelize SQL sub-plans across nodes. G-SQL [54] takes a similar approach, splitting queries between SQL Server and an in-memory graph engine. Every G-SQL query is semantically equivalent to a SQL query; the system dispatches graph patterns to the in-memory engine for performance, not because the operations require a different computational model. A dynamic-programming-based cost model optimizes the interleaving order of cut-set joins between engines, but the boundary itself is a syntactic marker, not an expressibility constraint.

The architecture of both systems demonstrates that graph queries benefit from combining declarative and imperative execution within one query. The surface resemblance is direct: G-SPARQL's retrieval-based operators map to PostgreSQL, its traversal-based operators map to the interpreter. The similarity is superficial. G-SPARQL's operator families are fixed, with no overlap in capabilities and no cost-based algorithm that searches over alternative assignments. G-SQL's cost model optimizes join interleaving across engines but does not govern the boundary itself, which remains structurally predetermined. Neither system exposes a backend-agnostic typed IR whose type discipline participates in placement or transfer-cost reasoning. Consequently, neither architecture generalizes to arbitrary k backends: adding a third computational model would require a third operator family or a third syntactic partition, not a solver that distributes existing operations across whichever backends can express them.

Prior work therefore shows that graph queries benefit from hybrid execution, and that a fixed split between declarative and imperative operators already yields quantifiable gains. No surveyed system, however, performs general, cost-based placement of a typed graph-query IR across backends with different computational models within a single query, at a granularity finer than the relational operator. These two components, a compilation infrastructure that preserves types for multi-target lowering (Chapter 3.2) and a placement mechanism that operates at the granularity of individual operations (Chapter 3.3), are addressed in turn.

3.2. Query Compilation and Intermediate Representations

Compilation for a single execution target is a well-studied problem with mature solutions. Multi-level IR architectures, SSA-form program representations, and ANF-based progressive lowering have each demonstrated significant speedups over interpreted query execution. HashQL's compilation pipeline builds on these foundations: the HIR is normalized into ANF, lowered to an SSA-form MIR, and optimized through a series of transformation passes (Chapter 5, Chapter 6). The pipeline is deliberately conventional: the stages that precede placement reuse proven infrastructure. The divergence is in what follows: rather than lowering the entire IR toward a single backend, placement partitions the MIR across backends with different computational models, and each partition is lowered independently (Chapter 7, Chapter 8).

MLIR provides compiler infrastructure within the LLVM project in which domain-specific dialects, each with its own types and operations, coexist in a single IR and are progressively lowered through pattern-based rewrites [49]. Michael Jungmair, et al. [36] apply this infrastructure to database query compilation with LingoDB. LingoDB defines four dialects for data processing: `relalg` for relational operators, `db` for database-specific scalar types, `dsa` for data structures and algorithms, and `util` for composite types absent from standard MLIR dialects. Queries are optimized through passes that implement selection pushdown, join reordering, and cross-domain optimization, then progressively lowered until the `llvm` dialect is reached. LingoDB validates that a multi-level open IR architecture reduces implementation effort while achieving competitive performance; the dialect pipeline is, however, single-target throughout.

LingoDB's dialect pipeline converges on a single target: all dialects ultimately lower to the `llvm` dialect and from there to native machine code. Heterogeneity is at the input level: domain-specific dialects allow SQL and PyTorch models to coexist in one query, but all fragments compile to the same execution environment. Convergence on LLVM provides an established optimization infrastructure, a substantial benefit that a custom IR forgoes. HashQL's heterogeneity is at the output level: a single IR must be partitioned across backends with different computational models, which changes what role types must play beyond compilation. In LingoDB, types drive progressive lowering within one pipeline; in HashQL, types are consumed across a partition boundary to determine backend eligibility and transfer cost (Chapter 7), serialization format and encoding ambiguity (Chapter 8.2), and direct value interpretation without lowering (Chapter 8.1).

Without type preservation across this boundary, neither placement nor serialization decisions can be made.

Timo Kersten, et al. [40] present Umbra, a compiled query engine built on a custom SSA IR purpose-built for database workloads. HashQL's MIR follows the same principle: a domain-specific IR with problem-specific constructs, in this case graph-effect terminators that model traversal operations as control-flow boundaries (Chapter 6.1). Umbra lowers relational plans to its IR in a single pass through the Tidy Tuples framework. Unlike LingoDB, Umbra targets two compilation backends: Flying Start for low-latency x86 emission and LLVM for peak throughput. Adaptive execution switches between them at run time in a just-in-time compilation (JIT)-like model: both backends produce machine code for the same CPU, and each can compile any query fragment. HashQL's backends each express only a subset of the MIR, and assignments are disjoint: the cost model and capability constraints that determine where each fragment executes operate on the IR directly (Chapter 7.2).

ANF has been established as a tractable intermediate form for query compilation. Amir Shaikhha, et al. [75] use ANF as the IR at every level of a multi-stage DSL stack, from query plans down to generated C. Ruby Y. Tahboub, et al. [77] show that the same architecture can be realized in a single generation pass via the first Futamura projection: specializing a staged interpreter with respect to a concrete query produces compiled code without explicit multi-pass lowering. HashQL uses ANF at the HIR level (Chapter 5): the naming discipline and explicit evaluation order that normalization provides are what make the transition to SSA-form MIR mechanical.

Other compilation systems share elements of this infrastructure. Poseidon compiles graph queries through LLVM with graph-specific operators; an adaptive interpreter serves as fallback during compilation [9]. Babelfish unifies relational operators and polyglot UDFs in a multi-level IR that fuses across language boundaries [27]. Raqlet compiles SQL, SQL/PGQ, GQL, and Datalog through a three-stage IR pipeline for source-language interoperability [76]. In each case, execution targets a single engine.

HashQL's PostgreSQL backend must compile MIR fragments into declarative SQL (Chapter 8.2). The MIR contains control flow that SQL cannot express directly: branches and conditionals have no native equivalent, and loops are expressible only through recursive common table expressions. Two lines of work address this compilation problem: structural compilation of control flow into SQL constructs, and pattern-based elimination of imperative code through relational transformation.

Tim Fischer, et al. [23] present a compilation strategy that maps CFG basic blocks directly into SQL common table expressions. Straight-line and branching control flow becomes non-recursive common table expression (CTE) chains; loops are encoded through a single recursive CTE whose working table carries variable bindings between iterations. This builds on earlier work by Denis Hirn and Torsten Grust [29], who compile PL/SQL through SSA and ANF into WITH RECURSIVE queries, and Denis Hirn and Torsten Grust [30], who refine the approach using trampolined style to handle arbitrarily nested iteration. Their work demonstrates that SSA and ANF are effective intermediate forms for compiling control flow to SQL. HashQL shares this conclusion: the HIR is normalized into ANF before MIR construction, and the MIR's SSA structure is what makes SQL code

generation feasible. Chapter 8.2 discusses the relationship between this approach and the strategy this work adopts.

Two alternative approaches address the procedural escape problem without compiling general control flow. Froid transforms imperative SQL UDFs into relational algebraic expressions using the Apply operator, pattern-matching recognized constructs such as variable declarations, assignments, and conditionals [67]. QBS takes a synthesis-based approach: it uses Z3 to infer loop invariants and postconditions in a theory of ordered relations, discovering equivalent SQL for imperative Java ORM code [15]. Both operate within a single RDBMS and target source-level constructs rather than a general IR, which makes them complementary to, rather than alternatives for, compilation from SSA form.

The compilation techniques reviewed here, multi-level IRs, SSA-form lowering, ANF normalization, and control-flow-to-SQL translation, are individually established. What remains open is their integration into a single typed IR designed for multi-target partitioning rather than single-backend lowering. The IR design (Chapter 6), the type-driven placement mechanism (Chapter 7), and the backend implementations (Chapter 8) address this gap.

3.3. Heterogeneous Work Placement

Assigning query fragments to execution backends requires decisions along three axes: the granularity of the placement unit, the capability model that determines which backends can execute which operations, and the transfer cost model that governs data movement between backends. Polystore and federated systems have explored this space from table-level routing through operator-level assignment, with cost models that capture bandwidth and compute throughput. HashQL's placement performs statement-level analysis to determine eligibility and cost, then assigns basic blocks that have been split into uniform-support regions within a single MIR body, across backends with different computational models, under capability constraints that propagate through data dependencies and transfer costs that account for type-dependent serialization (Chapter 7).

Tomas Karnagel, et al. [37] introduce an adaptive placement approach for heterogeneous CPU-GPU query processing that partitions a physical query execution plan into disjoint execution islands at compile time, delimited by estimation breakers where intermediate cardinalities become known. Placement optimization is then performed per island at run time with exact cardinality information, rather than globally over the full plan with estimated cardinalities. The approach operates at sub-operator granularity: relational operators are decomposed into reusable sub-operators (preprocessing, main computation, postprocessing phases) that can be placed independently on different compute units. The central empirical finding is that island-based regional optimization is robust to cardinality estimation errors. Global optimization over the full plan degrades when estimates are inaccurate; regional optimization within islands is unaffected, because it operates on cardinalities that are precisely known at island boundaries, and matches or exceeds global performance under realistic conditions, a finding that supports the regional decomposition adopted in Chapter 7.5.

HashQL's island concept is directly influenced by this work. The underlying placement problem is, however, different in kind. Because Karnagel's compute units share a common operator model via OpenCL, any unit can execute any kernel; the placement decision is purely a cost trade-off, and islands can be re-optimized at run time as new cardinality information becomes available. HashQL's backends have different computational models, where each can express only a subset of the MIR. Feasibility must therefore be determined before cost, and because backends require structurally different output, islands are compiled into backend-specific artifacts at compile time rather than re-optimized at run time (Chapter 7.2, Chapter 8).

Sebastian Kruse, et al. [47] present Rheem, a cross-platform optimizer that assigns dataflow operators to heterogeneous backends such as Spark, PostgreSQL, and Java Streams via cost-based plan enumeration. Each platform-agnostic operator is inflated with all platform-specific execution alternatives through graph-based operator mappings, which determine both the set of eligible platforms and their execution cost. Data movement between platforms is planned through a channel conversion graph, where each inter-platform conversion is an explicit cost-bearing operator; a minimum conversion tree algorithm selects the lowest-cost movement plan for each transition. When observed cardinalities diverge from estimates during execution, progressive re-optimization adjusts the remaining plan at inserted checkpoints.

Rheem demonstrates that automatic cross-platform placement yields order-of-magnitude gains over single-platform execution, and its channel conversion graph makes inter-platform data movement planning explicit. The placement problem it solves is, however, structurally different. Rheem maps self-contained dataflow operators to platforms: each operator is independently eligible or not, and transfer cost is a property of the conversion path between platforms. HashQL performs statement-level eligibility analysis within a single function body, where support is not independent: an unsupported operation renders all downstream consumers ineligible on that target, because the target cannot produce the intermediate value (Chapter 7.2). Transfer cost is correspondingly finer-grained: rather than a per-conversion resource cost, it accounts for type-dependent serialization overhead and the origin-specific cost of accessing vertex data stored on a different backend (Chapter 7.1).

Jennie Duggan, et al. [21] introduce BigDAWG, a polystore that organizes heterogeneous backends into data-model islands: a relational island for SQL, an array island for array processing, and a text island for document retrieval. Each island maintains its own query language, optimizer, and execution engine, with cross-island queries decomposed at subquery boundaries. The island concept is data-model-scoped: all relational operations belong to the relational island regardless of which backend executes them. This scoping is the fundamental divergence. HashQL's islands are execution-target-scoped: operations within the same data model are partitioned across backends because different backends can express different subsets of the MIR. Because BigDAWG maintains no single IR across islands, there is no representation in which an operation-level cost model or capability constraint could be expressed.

Other systems span the granularity spectrum without reaching operation level. PolyBase performs a binary split between SQL Server and Hadoop, pushing selective

operators to MapReduce based on selectivity estimates [20]. MuSQLE extends this to subquery-level multi-engine analytics across PostgreSQL, MemSQL, and SparkSQL, where engines are interchangeable for SQL and the choice is performance-driven [25]. Bobbi W. Yogatama, et al. [81] go below operator granularity with segment-level query plans for heterogeneous CPU-GPU execution, where different data segments of the same column execute different plans depending on segment residency; the placement is data-residency-driven rather than capability-constrained. At the other end of the historical spectrum, S. M. Deen [19] address the capability constraint directly: PRECI* decomposes queries for heterogeneous distributed databases where not all nodes can perform all operations, an early example of feasibility-constrained heterogeneous decomposition, and the property it recognizes is central to Chapter 7.2.

Transfer cost modeling across these systems follows a consistent pattern. The foundational distributed cost model decomposes query cost into CPU, I/O, and per-byte communication terms [52]. Rheem's channel conversion graph extends this with explicit conversion operators, but the per-conversion cost remains resource-based: a function of CPU, memory, network, and disk utilization. Karnagel's model captures compute throughput and PCIe bandwidth between compute units that share a memory address space. In geo-distributed settings, the overhead of transferring intermediate results between engines has been observed to account for a substantial fraction of total cost [31], consistent with the prominence of transition cost in HashQL's cost model (Chapter 7.3). These models vary in sophistication, from linear per-byte terms to Rheem's resource-based conversion costs, but the transfer component remains volume-proportional: cost scales with aggregate data size, without dependence on the types of the values being moved.

HashQL's cost model derives size estimates from the MIR types of each transferred local, so different values contribute different costs depending on their type structure. This type-level granularity serves two roles absent from the surveyed literature. First, types participate in eligibility: values whose representation is ambiguous under the target's type system are rejected outright, a capability constraint that volume-based models cannot express (Chapter 7.2). Second, the cost model distinguishes graph-sourced data from program-produced locals. Locals flow along CFG edges and are costed at transitions; vertex path data is fetched directly from origin to consumer, bypassing intermediate backends, and is costed as a per-block premium rather than an edge cost (Chapter 7.1).

HashQL builds on this foundation: its islands derive from Karnagel's regional optimization, and its cost-based objective follows the same principle as Rheem's cross-platform enumeration. The specific configuration has not been addressed: statement-level eligibility analysis within a typed IR, with block-level assignment after splitting, where capability constraints propagate through data dependencies and the cost model distinguishes graph-sourced data from program-produced locals. Table 7 summarizes the surveyed systems against these properties. Together with the compilation infrastructure and graph execution model surveyed above, this constitutes the pipeline presented in Chapter 7 through Chapter 8.

4. System Overview

The HashQL compiler follows the pipeline structure common to compiled, statically typed languages: the same program is lowered through a series of intermediate representations. Each representation removes surface-level complexity while formalizing and simplifying structures that subsequent transformation passes depend on. The compiler is frontend-agnostic by design and exposes a functional, referentially transparent language. This allows different surface syntaxes to map onto the compiler, either directly or indirectly [3].

The currently available frontend, J-Expr, was inspired by S-expressions in LISP. Expressions are parsed into an AST, which operates at the syntactic level. It resolves imports and items to their absolute counterparts, and formalizes special forms into dedicated nodes. The AST is then lowered into the HIR, which preserves the tree-like shape while removing syntactic sugar. The HIR serves as the basis for type inference and type checking, as well as initial optimization steps that are more naturally expressed on a tree-like structure – such as hoisting – than on the statement-based structure present in the MIR. The AST and HIR were developed as part of the Großer Beleg; the remainder of the pipeline is the contribution of this work [11, 57].

This work extends the existing pipeline toward heterogeneous execution. This work introduces a new intermediate representation, the MIR: a block-structured, type-preserving, SSA IR. To facilitate this transition, the HIR is first normalized into administrative normal form. Compilers such as Rust or Swift lower through LLVM IR to a single machine target; the HashQL pipeline instead performs its analysis and optimization at the MIR level, then lowers each assigned region to a target-specific representation. Placement decisions require a representation that is statement-based rather than tree-shaped, yet remains target-independent: committing to a single execution model at this stage would foreclose the ability to assign individual operations to different backends. Graph effects, operations that traverse or mutate the underlying bi-temporal graph as discussed in the Großer Beleg, require dedicated representation in the IR. We chose a custom IR over existing alternatives, because it allows these effects to be modeled as terminators. This allows MIR passes and execution analysis to treat graph operations as ordinary control-flow boundaries, rather than requiring special handling [11, 48].

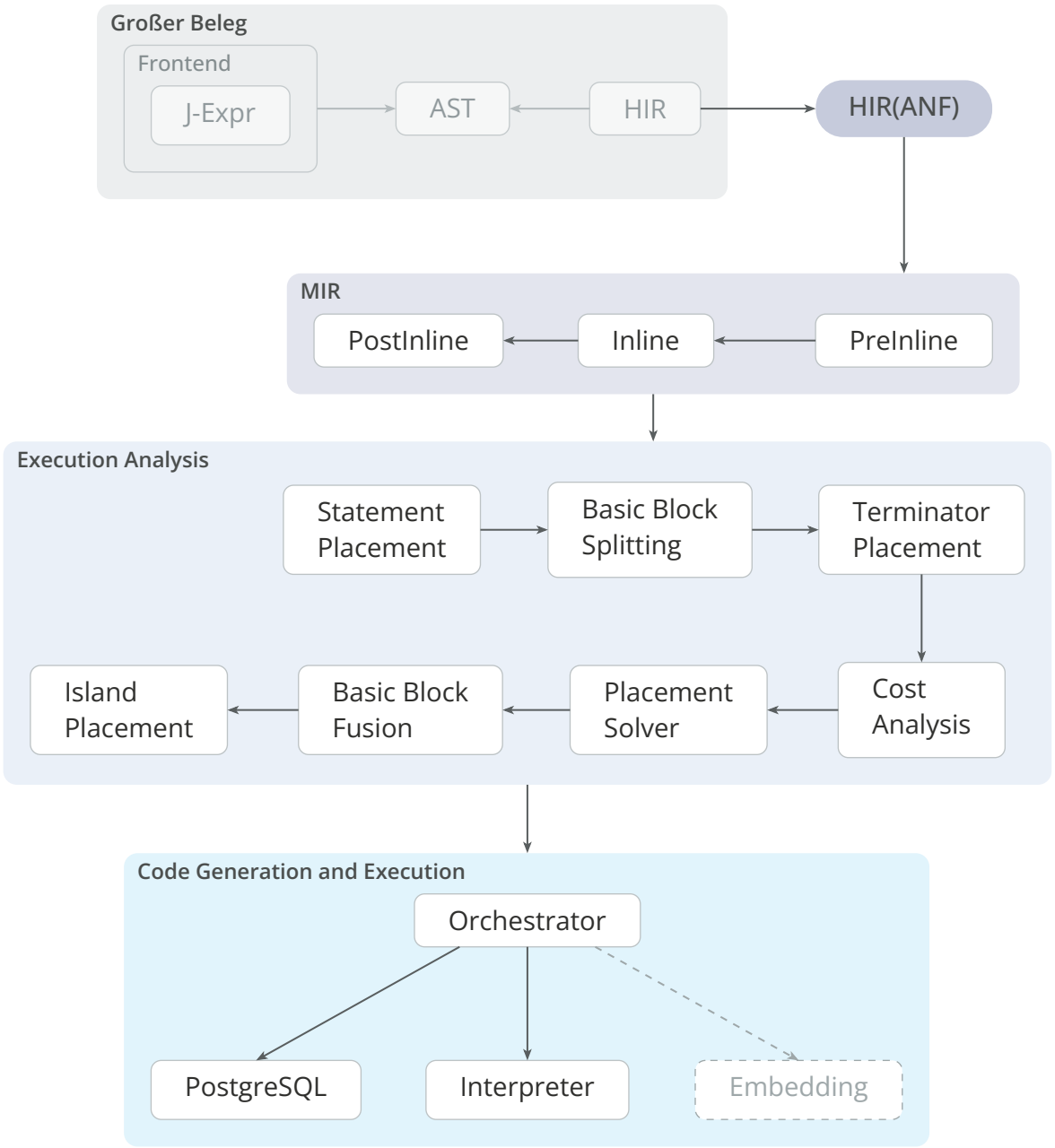


Figure 1: Compiler Pipeline

Figure 1 illustrates the full pipeline. The stages from the high-level intermediate representation in administrative normal form (HIR/ANF) onward consist of three distinct phases. First, the MIR undergoes a series of transformation passes that prepare the program body for analysis. The central transformation is inlining, which expands the body into a form the execution analysis can operate on. Passes before and after inlining reduce complexity and normalize the control-flow graph. Second, an execution analysis determines where each operation should run: it computes per-statement costs, splits basic blocks to allow uniform scheduling, solves for a minimum-cost assignment, and fuses blocks that were split but received the same assignment back together. The analysis then groups the resulting blocks into execution islands, supplemented by data islands that source auxiliary information which cannot be produced by preceding islands. Third, the orchestrator dispatches each island to its backend and marshals data

between them. This work targets three execution backends: PostgreSQL, which compiles MIR fragments into SQL queries; a runtime interpreter, which serves as a universal fallback; and an embedding backend for vector operations, which is supported during analysis but not yet connected to an execution engine.

4.1. Mid-Level Intermediate Representation

The MIR is modeled after existing compiler intermediate representations, with the primary influences being the Rust MIR, Swift intermediate language (SIL), MLIR, and LLVM IR. A MIR program consists of bodies. Each body is modeled as a control flow graph, containing one or more basic blocks, the first of which is the entrypoint. A basic block is the atomic unit of the CFG: a sequence of statements executed unconditionally, terminated by a single terminator. Terminators produce the edges of the CFG and may either return to the caller, transfer control to another basic block, or suspend execution to yield to the orchestrator [26, 48, 56].

The MIR is in SSA form, meaning that each local is assigned exactly once. Because every use of a value has a unique definition site, dataflow analysis over the MIR is straightforward. Unlike LLVM IR, which uses φ nodes to merge values at join points, the HashQL MIR uses block parameters. Block parameters map naturally to the functional language exposed in earlier stages and make dataflow between blocks explicit. Andrew W. Appel [5] shows that φ nodes and block parameters can be used interchangeably without loss of expressiveness [18, 49].

Terminators serve two distinct roles. The first is classical control flow, in which a terminator either transfers control to another basic block or returns to the caller. The second is executing effects, in which the terminator yields to the underlying orchestrator. An effect is a coroutine that yields its continuation not to the caller, but directly to the effect system. Because continuation passing is inherently a control-flow operation, modeling effects as terminators allows transformations and analyses to treat them uniformly, without special cases [39, 64].

The MIR also preserves type information, where each local carries the type determined during HIR type checking or derived during MIR transformation. This type information guides the placement decision and backend translation passes. In addition to the explicit CFG, a dataflow analysis framework exposes an implicit dataflow graph. This enables analyses at the data level, such as liveness analysis, which is used throughout the pipeline for SSA repair and terminator cost determination [12].

4.2. HIR Normalization

The HIR and the MIR serve fundamentally different purposes. The HIR preserves the programmer's intent, permitting nested expressions, implicit closure environment capture, and lazy global references. The MIR instead encodes program semantics: the result of any operation must be stored in a variable, every function reference must be explicit, and closures carry their captured variables as concrete data. This difference in program representation requires a non-trivial normalization step.

First, the HIR is converted into administrative normal form. ANF occupies a middle ground between the two: it preserves the tree structure of the HIR while imposing the naming discipline of the MIR – every subexpression is bound through a let-binding and evaluation order is explicit. This is closely related to continuation-passing style; Cormac Flanagan, et al. [24] shows that the two can be converted into each other without loss of expressiveness. Continuation-passing style (CPS) maps more cleanly to function-calling conventions, but because the HIR already uses dedicated nodes for operations such as comparisons, we did not pursue CPS. ANF instead operates on the existing tree, making the conversion straightforward [24, 63, 72].

Unlike systems programming languages, such as Rust, which differentiate between closures and function pointers, HashQL makes no attempt at differentiation. Every function is a closure, and may implicitly capture an environment: variables that are mentioned in the body, but defined in an outer scope. Because the MIR encodes program semantics, it must track all live variables and therefore does not support implicit closure captures. To make the transition between HIR and MIR possible, the reification step constructs an explicit environment variable for each closure, which is passed as an argument on every invocation (Chapter 5.5).

The HIR also permits module-level definitions in a script-like fashion, with no explicit entrypoint function. This permits scripting-style usage and simplifies module composition. One design goal of the MIR is that every body must be invocable explicitly. To convert module-level definitions and global constants into bodies, a technique called thunking wraps each global definition in a deferred call. This is sound because the language is referentially transparent and immutable: any execution of the same expression under the same circumstances must produce the same result [32, 71].

Once normalized, the high-level intermediate representation in administrative normal form (HIR/ANF) is reified into the MIR. Because ANF already names every subexpression and fixes evaluation order, most of this translation is mechanical; terminator translation and block wiring are the most involved steps. The resulting MIR body exposes every operation as an individually addressable statement with explicit data dependencies, which is the granularity that execution analysis requires [5, 39].

4.3. Target Placement

The underlying bi-temporal graph is heterogeneous, with data pertaining to entities stored across different execution targets. Minimizing the data transferred between targets is critical, as each transfer incurs serialization, deserialization, and bandwidth overhead. Colocating operations within the same target reduces this cost, yet this exposes a fundamental tension: the most efficient target for an operation is not always capable of running it, and grouping operations to minimize transfers may require sacrificing the efficiency of each. The placement is therefore governed by three concerns: correctness, as each backend can only express a subset of the language; execution cost, as backends differ in how efficiently they evaluate the same operation; and transfer cost, as moving data between backends is not free. Because costs differ depending on the backend chosen and because programs in the MIR may contain loops and branching, finding an optimal assignment is a combinatorial optimization problem. This

work proposes a solver that uses a greedy double-pass algorithm, with branch-and-bound for strongly connected regions, to minimize a cost surrogate that accounts for both computation and transfer expenses (Chapter 7.5) [42].

The analysis proceeds in three phases: preparation, solving, and mending. During preparation, each statement and terminator is attributed a per-target cost depending on whether it can be scheduled on a given backend. Once these costs are derived, the existing basic blocks are split so that each block contains only operations eligible for the same set of backends. Finally, each possible target-to-target permutation of each transition is assigned a cost consisting of a transfer cost derived from the data that must cross the target boundary and an inherent transition overhead, while each block additionally receives a vertex path premium reflecting the cost of accessing data from remote origins (Chapter 7.1). The solving phase then assigns each block to a single target while minimizing cost. While the solver produces a valid assignment, the segmented basic blocks created during splitting are inefficient for execution. The mending phase therefore fuses chains of blocks that have been assigned to the same backend, significantly reducing the number of terminators and the overhead they introduce. The resulting blocks are then grouped by target into islands, which are scheduled for dispatch by the orchestrator.

The execution analysis produces a set of islands. An execution island contains both the operations to be executed and the data they require. A data island exists solely to retrieve data. If a preceding execution island already produces the required data, it is passed forward; otherwise, a data island is created to fetch it. Because data islands do not participate in control flow and their requirements are statically determined, they can be batched and parallelized at the start of execution. Data islands are always a last resort, as they add overhead that can be avoided in most cases. The solver minimizes the cost surrogate, which naturally colocates operations with the data they operate on.

4.4. Code Generation and Execution

Execution proceeds in two phases: compilation and dispatch. During compilation, each backend compiles across all graph effect operations at once rather than treating each in isolation. For the PostgreSQL backend, this means taking every island across every effect operation and compiling them into a single prepared statement with its required parameters. The interpreter, by contrast, has no separate artifact and operates directly on the existing MIR. Splitting compilation from dispatch enables caching: the compiled artifacts, together with the island graph, contain enough information to dispatch a query without re-running the compilation pipeline.

The execution loop is coordinated by the orchestrator, with the interpreter driving program execution by tree-walking the MIR directly. When the interpreter encounters an effect terminator, such as a graph read operation, it yields a suspension to the orchestrator, which fulfills the request and upon completion resumes the interpreter with the supplied continuation value. Nested graph effects are handled by recursing into the orchestrator upon encountering a suspension. This is possible because the execution analysis constrains effects to the interpreter.

5. HIR in Administrative Normal Form

The HIR is a tree-structured representation common to early stages of compilers for functional languages: expressions nest freely, closures capture variables implicitly, and evaluation order is determined by the tree's structure rather than stated explicitly. While this structure facilitates optimizations that are more naturally expressed on trees, it impedes the per-statement analysis that placement and backend translation require.

The transformation from HIR to MIR cannot proceed directly, because the HIR lacks the naming discipline and explicit evaluation order that the MIR requires. An intermediate representation that this work calls high-level intermediate representation in administrative normal form (HIR/ANF) bridges this gap. The transformation from the type-checked HIR to the HIR/ANF proceeds through four steps:

1. **Specialization.** The specialization pass, initially introduced in the Großer Beleg, receives a minor extension in this work to assemble graph operation chains as first-class `GraphRead` nodes, which are then translated into graph effect terminators (Chapter 6.1) [11].
2. **Normalization.** Converts the HIR into ANF by binding every computation to a fresh variable and partitioning the tree into scoped boundaries [24, 72].
3. **Effect Hoisting.** Promotes bindings from inside effectful closures to the enclosing scope when they do not depend on the closure's parameters, a form of loop invariant code motion that reduces per-entity computation and improves placement granularity.
4. **Thunking.** Wraps every module-level definition in a deferred call, breaking the ANF property. The pass converts script-style local computations into explicitly invocable bodies that the MIR requires.

Because thunking breaks the ANF property, normalization runs a second time to restore the invariant. Once complete, the HIR/ANF is reified into the MIR. Because ANF and SSA share the same naming discipline, this translation is largely mechanical: let-bindings become assignments, atoms become operands, closures become bodies with explicit environment capture, branches become `SwitchInt` terminators, and graph operations become `GraphRead` terminators [5, 59].

5.1. ANF

HIR/ANF enforces the administrative normal form (ANF) property over the HIR. ANF distinguishes between two classes of expressions: atoms, which denote values that

require no further evaluation, and computations, which produce values through operations on atoms. Every computation is bound to a fresh variable through a let-binding, which explicitly names the intermediate result and converts the tree-shaped nesting into a flat evaluation sequence. The evaluation order is fully determined by the binding position. This naming discipline corresponds directly to the SSA property of the MIR: each let-binding maps to exactly one MIR assignment, and each atom maps to an operand. Like the SSA property, the ANF property can be broken by transformations and must subsequently be repaired [5, 24, 72].

$p ::=$	primitives:	$c ::=$	computations:
$\mid n \in \mathbb{Z}$	integer	$\mid a\ a$	application
$\mid r \in \mathbb{R}$	float	$\mid a_1 \circ a_2$	binary operation
$\mid s$	string	$\mid \circ a$	unary operation
$\mid \text{true} \mid \text{false}$	boolean	$\mid \langle i_1 : a_1, ..., i_n : a_n \rangle$	struct
$\mid \text{null}$	null	$\mid (a_0, ..., a_{n-1})$	tuple
$x ::=$	variables:	$\mid [a_1, ..., a_n]$	list
$\mid l$	local	$\mid \{a_1 : a_{1'}, ..., a_n : a_{n'}\}$	dict
$\mid q$	qualified	$\mid \text{input}(x)$	input operation
$f ::=$	fields:	$\mid T$	type constructor
$\mid n \in \mathbb{N}$	positional	$\mid \lambda(x_1 : T_1, ..., x_n : T_n).b$	closure (boundary)
$\mid i \in \mathcal{I}$	named	$\mid \text{thunk}\ b$	thunk (boundary)
$\tau ::=$	graph tail:	$\mid \text{if}\ a\ \text{then}\ b_1\ \text{else}\ b_2$	conditional (boundary)
$\mid \text{collect}$	collect	$\mid \text{graph}(a, c_1, ..., c_n, \tau)$	graph operation
$a ::=$	atoms:	$b ::=$	boundaries:
$\mid p$	primitive literal	$\mid (\text{let}\ l_1 = c, ..., l_n = c_n)a$	binding sequence
$\mid x$	variable		
$\mid (.\ f\ a)$	field projection		
$\mid ([]\ x\ a)$	index subscription		

Figure 2: ANF Syntax

Figure 2 defines the grammar for a valid ANF over the type-checked HIR. Primitives are the literal values present in the program and are classified as atoms, because they require no further evaluation. Variables are also atoms and come in two forms: local variables, which reference a binding in the current or an enclosing scope, and qualified variables, which reference a binding by its fully resolved path. The HIR pipeline presented in the Großer Beleg eliminates all imports and resolves them to qualified names, which permits a strict separation of the two forms [11].

Traditional ANF classifies only primitives and variables as atoms Cormac Flanagan, et al. [24]. This work additionally classifies field projections (static access to struct or tuple members) and index subscript operations (dynamic access to list or dictionary elements) as atoms, provided their target is itself an atom. For index subscript operations, the grammar further requires the subscript to be a variable rather than an

arbitrary atom, which ensures the index value is bound before the subscript operation is evaluated. This deviation is justified by the MIR's place model, which natively supports projection chains as first-class constructs; flattening these chains during normalization would introduce intermediate let-bindings that subsequent MIR passes would eliminate. The HIR outside ANF permits projections and subscript operations on arbitrary sub-expressions; under ANF, the target must itself be an atom. Because all three atom categories are free of nested computations, the normalization pass can treat them uniformly as operand positions that require no further extraction.

Computations cover all forms that require evaluation. The first set is analogous to traditional function application: function calls, input operations, and binary and unary operations all consume atoms and produce a single result. The second set covers aggregate construction: structs, tuples, lists, and dicts each assemble multiple atoms into a composite value. While aggregates may not appear to involve computation, they produce a new value from their operands and therefore require their arguments to be atoms. The final non-boundary computation is the type constructor: a compiler primitive that generates a closure accepting the inner type of a nominal newtype and producing a value of that type. Because a type constructor is equivalent to a closure, it cannot be an atom.

Four computation forms cannot be atoms: closures, thunks, conditionals, and graph operations. Each contains nested bodies with their own bindings and evaluation sequences. To accommodate nested evaluation, these forms introduce boundaries. A boundary is a scoping construct: a new binding scope consisting of a sequence of let-bindings followed by a single atom that serves as its result. Bindings inside a boundary are invisible to the enclosing scope; only the trailing atom is exposed. Closures and thunks introduce boundaries, because their bodies are separate execution contexts: bindings inside a closure must not escape into the enclosing scope, as they may reference the closure's parameters, which do not exist in the enclosing scope. Conditionals are natural boundaries: only one arm is evaluated at run time, so promoting a binding out of a branch would cause unconditional evaluation, which changes program semantics. Graph operations introduce boundaries to separate effectful computations over a remote execution backend from the pure evaluation context of the enclosing scope.

5.2. Normalization

Normalization converts a type-checked HIR, which may contain arbitrarily nested computations in operand positions, into a well-formed HIR/ANF. The pass is analogous to the SSA repair step presented in Chapter 6.3.2: both enforce a structural invariant that the preceding stage may have violated, and both are idempotent.

The ANF grammar requires that every operand position contains an atom; computations may only appear as the value of a `let` binding. Normalization enforces this constraint through a technique this work calls let-floating: each sub-expression is normalized, and any result that is not an atom is bound to a fresh local in the current boundary's binding list; the original sub-expression is replaced with a reference to that local. Because children are processed before their parent, bindings accumulate in

evaluation order. When a boundary closes, the accumulated bindings and the trailing atom form a single `let` expression.

Algorithm 1: ANF Normalization

```
1:  procedure Normalize(node, bindings)
2:      for each child c of node do
3:          if node requires boundary for c then
4:              c ← Boundary(c)
5:          else
6:              c ← Normalize(c, bindings)
7:              c ← EnsureAtom(c, bindings)
8:          end
9:      end
10:     return node
11: end
12:
13: procedure Boundary(node)
14:     bindings ← empty list
15:     node ← Normalize(node, bindings)
16:     node ← EnsureAtom(node, bindings)
17:     if bindings ≠ ∅ then
18:         return let bindings in node
19:     end
20:     return node
21: end
22:
23: procedure EnsureAtom(node, bindings)
24:     if node is an atom then
25:         return node
26:     end
27:     l ← fresh local
28:     append l = node to bindings
29:     return l
30: end
```

Algorithm 1 presents the core algorithm as three procedures. `Normalize` recurses into a node's children and distinguishes two kinds. Children that introduce a separate evaluation context (closure bodies, thunk bodies, conditional arms, and short-circuit right operands, as outlined in Figure 2) are processed through `Boundary`, which opens a fresh binding scope. All other children are normalized within the current boundary and passed to `EnsureAtom`. `EnsureAtom` is the classification step: atoms pass through unchanged; computations are bound to a fresh local and replaced by a reference to it.

The ordering invariant follows from the algorithm's recursive, child-first design: a binding is appended only after its value has been fully normalized, so inner computations always precede outer ones in the list. This converts an arbitrarily deep nesting of expressions into a flat ANF binding sequence. Four aspects require separate treatment: nested let-bindings, type assertions, short-circuit boolean expressions, and projection chains.

Let Flattening. The HIR permits nested `let` expressions: a binding's value may itself contain a `let`. When normalization encounters a `let` node, each binding's value is normalized and appended directly to the current scope's binding list. As Algorithm 2

shows, `EnsureAtom` is not called on the value: the normalized computation remains as the binding's value without additional indirection. The `let` node itself is dissolved; its body replaces it and continues through normalization.

Algorithm 2: Let flattening

```
1: procedure NormalizeLet({(l_1, v_1), ..., (l_n, v_n)}, body, bindings)
2:     for each binding (l_i, v_i) do
3:         v_i ← Normalize(v_i, bindings)
4:         append l_i = v_i to bindings
5:     end
6:     ▷ The let node is dissolved; continue with the body
7:     return Normalize(body, bindings)
8: end
```

Because values are processed before their bindings are appended, inner bindings always precede outer ones, preserving evaluation order. This flattening is essential in the presence of ANF: every non-atomic sub-expression introduces a `let` binding, so without dissolution the tree depth would grow proportionally to expression complexity.

Type Assertion Erasure. Type assertions narrow types and guide inference during type checking. Because normalization operates on type-checked HIR, where every type has already been inferred and monomorphized, these assertions carry no further semantic value and have no run-time representation. Normalization removes them by replacing each assertion node with its underlying value, as outlined in Algorithm 3. A type assertion wrapping an atom therefore evaluates to that atom directly, without introducing a binding [11:5.5.3].

Algorithm 3: Type assertion erasure

```
1: procedure NormalizeTypeAssertion(value, bindings)
2:     value ← Normalize(value, bindings)
3:     ▷ The assertion node is replaced by its underlying value
4:     return value
5: end
```

Short-Circuit Desugaring. Binary boolean operations (`&&`, `||`) implement short-circuit evaluation: for $A \wedge B$, B is only evaluated if A is true; for $A \vee B$, B is only evaluated if A is false. When the right operand is already an atom, short-circuiting is trivial, because there is no computation to defer, and the expression remains a binary operation. When the right operand is a computation, the operation is rewritten into its conditional equivalent.

$$A \wedge B \equiv \text{if } A \text{ then } B \text{ else false}$$
$$A \vee B \equiv \text{if } A \text{ then true else } B \tag{1}$$

A	B	$A \wedge B$	if A then B else false	$A \vee B$	if A then true else B
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1

Table 1: Short-circuit desugaring equivalence

Type checking guarantees that both A and B evaluate to boolean values. Under this constraint, the equivalence presented in Table 1 holds, as confirmed by exhaustive enumeration: the only value that short-circuits $A \wedge B$ is $A = \text{false}$, which is exactly the value the else branch returns, and dually for $A \vee B$. This equivalence determines the structure of the algorithm outlined in Algorithm 4: A is always evaluated, so it is normalized within the current boundary and ensured to be an atom. B is conditionally evaluated and therefore requires its own boundary.

Algorithm 4: Short-circuit boolean desugaring

```
1:  procedure NormalizeShortCircuit(op, a, b, bindings)
2:      a ← Normalize(a, bindings)
3:      a ← EnsureAtom(a, bindings)
4:      b ← Boundary(b)
5:      if b is an atom then
6:          ▷ No control flow needed
7:          return a op b
8:      end
9:      if op = ∧ then
10:         return if a then b else false
11:     end
12:     if op = ∨ then
13:         return if a then true else b
14:     end
15: end
```

Projection Chains. Field and index access expressions form projection chains, which the MIR lowers to place expressions with projection sequences (Chapter 6.1). Projections share properties of both atoms and computations: unlike other atoms, they have children (Figure 2), but unlike computations, they do not produce a new value. Because the MIR represents projections as first-class components of place expressions, they are equivalent to atoms; binding them to intermediate locals would inflate the MIR with bindings that subsequent passes immediately eliminate.

Normalization therefore enforces two constraints: the children of a projection must themselves be valid projections or locals, and it must avoid generating unnecessary bindings for expressions that are already projections. Type checking rejects projections over primitive values, so the base of a projection is never a literal. If the base is a computation, `EnsureProjection` binds it to a fresh local; after normalization, the base is therefore always a variable.

Index subscript operations impose a stricter constraint: the index must be a local variable rather than any atom. While the ANF grammar permits any variable in this position, qualified variables are converted to thunk calls during thunking (Chapter 5.4). Because thunk calls are computations, re-normalization binds them to locals. After the full pipeline, the index is therefore always a local, which guarantees that the index computation is bound before the subscript operation and that no nested projection chains exist within the index position. Algorithm 5 formalizes these procedures.

Algorithm 5: Projection normalization

```
1:  procedure EnsureProjection(node, bindings)
2:      if node is a variable or access expression then
3:          return node
4:      end
5:      l ← fresh local
6:      append l = node to bindings
7:      return l
8:  end
9:
10: procedure NormalizeFieldAccess(expr, field, bindings)
11:     expr ← EnsureProjection(expr, bindings)
12:     return expr . field
13: end
14:
15: procedure NormalizeIndexAccess(expr, index, bindings)
16:     expr ← EnsureProjection(expr, bindings)
17:     index ← EnsureLocal(index, bindings)
18:     return expr [index]
19: end
```

5.3. Effect Hoisting

Graph reads encapsulate effectful computations over the vertices of the underlying bi-temporal graph, following a map-reduce architecture as elaborated in the Großer Beleg. The closures inside a graph read pipeline are iteration bodies: each executes once per entity, forming a natural loop that may iterate thousands of times; any computation that can be hoisted out of a pipeline body and evaluated once yields two benefits: fewer statements for the interpreter to execute per iteration, and fewer statements for execution analysis to process [11].

Closures that are part of an effectful computation are lowered into bodies with provenance information, which informs the MIR that they participate in execution placement analysis and may be offloaded to an external execution target. Because placement operates at the granularity of these bodies, a binding eligible for hoisting, one that does not depend on any of the closure's parameters, can be separated from the offloaded body and evaluated in a cheaper context. If such a binding is not hoisted, two outcomes are possible: it is scheduled on a remote execution target, in which case any variables it references must be serialized at bandwidth cost and the computation is redundantly evaluated on the remote backend for each iteration despite being loop-invariant; or it remains on the interpreter, where it compounds per-iteration evaluation cost with-

out serialization overhead. A hoisted computation is instead evaluated in the parent context, which may itself reside inside an effectful terminator or in the host execution context on the interpreter.

The mechanism is similar to traditional loop invariant code motion (LICM), but unlike loop invariant code motion (LICM), which operates on CFG loops by relocating instructions to a loop preheader, effect hoisting achieves the same effect through scope promotion in the HIR/ANF, moving a let-binding from one boundary to the next enclosing boundary. Because the transformation is a simple tree traversal that relocates bindings between scopes, it preserves the ANF property [3:9.5].

Before

```
let %0 = entities(axis)                                    hir
        |> filter((v:0: Entity): Boolean ->
            let %1 = make_id("abc"),
                %2 = v:0.entity_id == %1
            in %2)
        |> collect
in %0
```

After

```
let %1 = make_id("abc"),                                   hir
    %0 = entities(axis)
        |> filter((v:0: Entity): Boolean ->
            let %2 = v:0.entity_id == %1
            in %2)
        |> collect
in %0
```

Listing 1: Effect hoisting: `%1` is promoted because it does not reference `v:0`

Listing 1 illustrates the effect. The binding `%1 = make_id("abc")` does not reference the closure parameter `v:0` and is promoted to the enclosing scope, where it is evaluated once. Without this promotion, the call would either be inlined (Chapter 6.3.1), increasing the amount of code that execution analysis must process, or if not inlined, force the complete filter expression to be evaluated on the interpreter, because function application cannot transition from the interpreter to PostgreSQL (Chapter 7.2). The comparison `%2 = v:0.entity_id == %1` references `v:0` and remains inside the closure.

Algorithm 6 formalizes the classification through a dependency infection model: for each binding, the algorithm computes the free variables of its value. If none are scope-dependent, the binding is promoted. Because the HIR does not support recursive bindings or loops, a single forward pass suffices. If a binding does depend on a scope variable, its output variable joins the scope-dependent set, therefore infecting downstream bindings that reference it.

Algorithm 6: Effect hoisting: binding analysis

procedure HoistBindings($\mathcal{B}, S$)
  ▷ Given bindings $\mathcal{B}$ and scope-dependent variables S
  for each binding (x, e) in $\mathcal{B}$ **do**
    $D \leftarrow \text{FreeVars}(e)$
    if $D \cap S = \varnothing$ **then**
      promote (x, e) to enclosing scope
    else
      $S \leftarrow S \cup \{x\}$
      retain (x, e) in current scope
    end
  end
end

Two restrictions preserve soundness. Hoisting activates only for closures nested inside graph operations; outside a graph context, closures are not iteration bodies, and promoting their bindings would change evaluation timing without reducing redundancy. Traditional LICM addresses non-loop hoisting through cost heuristics; this work does not, as graph iteration bodies provide a clear cost signal that non-iteration closures lack. Conditional expressions are also hoisting barriers: promoting a binding out of a branch would cause unconditional evaluation of code that the original program evaluates only when a specific branch is taken. For pure computations this introduces redundant work; for effectful computations, such as nested graph reads, it would alter program semantics.

This optimization is not universally beneficial: a purely constant expression such as $2 + 2$ would be folded to 4 by MIR instruction simplification (Chapter 6.3.2) inside the body of an effectful computation, and subsequently removed during dead store elimination after copy propagation, eliminating the need for a transfer. Hoisting is detrimental in this case, as it forces evaluation of the result in the parent context, where the constant must be transferred into the closure environment — a cost that would have been unnecessary otherwise. Cross-body constant propagation as outlined in Chapter 10.1.4.4 could recover these cases, but we reserve it for future work, as we consider the immediate benefit marginal.

5.4. Thunking

The MIR has no concept of files or global values and operates on the assumption that every body must be explicitly defined and invocable. By contrast, the HIR permits module-level definitions with no explicit entrypoint; bindings at the top level are values or closures. Thunking resolves this mismatch by wrapping each top-level binding in a thunk, a zero-argument closure with no environment capture, and replacing all references to those bindings with calls. The result is a program in which no top-level computation takes place directly; every top-level definition is accessed through callable indirection [32].

This transformation is sound because HashQL is referentially transparent and immutable: evaluating the same expression under the same environment always produces the same result, so deferring evaluation into a thunk does not change program semantics. The same guarantee holds for effectful computations: as Bilal Mahmoud [11:5.2] establishes, graph effects are safe to re-execute under the same temporal constraints, so thunking an effectful binding does not alter observable behavior. Bindings that are already thunks are therefore preserved unchanged: double-wrapping would introduce an incorrect extra level of indirection [71].

Before

```hir
let a:0 = 2,
    b:0 = 3,
    %0 = a:0 >= b:0
in %0
```

After

```hir
let a:0 = thunk -> 2,
    b:0 = thunk -> 3,
    %0 = thunk ->
        let %1 = a:0(),
            %2 = b:0(),
            %3 = %1 >= %2
        in %3
in %0
```

Listing 2: Thunking: top-level bindings wrapped, references become calls

Listing 2 illustrates the transformation. Each top-level binding (`a:0`, `b:0`, `%0`) is wrapped in a `thunk -> ...`, and references to `a:0` and `b:0` inside the body of `%0` become calls (`a:0()`, `b:0()`). As function application constitutes a computation, thunking is ANF breaking. The pipeline therefore runs normalization a second time immediately after thunking, restoring the ANF invariant before reification consumes it.

5.5. Reification

Reification translates the HIR/ANF into the MIR. Because ANF and SSA share the same naming discipline (Chapter 6), the core translation is mechanical: most computations become assignment statements whose rvalues correspond directly to their HIR counterparts. Binary, unary, and input operations map to their respective rvalue kinds; struct, tuple, list, and dict construction are unified into a single aggregate rvalue. Atoms are lowered to operands, with integers, booleans, and null values promoted to their non-opaque constant representation [5, 39].

Conditional expressions, presently limited to if-expressions (Chapter 10.1.3.1), are converted into `SwitchInt` terminators. Reification creates a join block with a single block parameter for the result value. Each branch is lowered into its own block sequence; once the branch body has been translated, it issues an unconditional `Goto` to the join block, passing its result atom as the block argument. Because the MIR has no notion of booleans, the discriminant is lowered to its integer equivalent: $\text{false} \to 0$ and $\text{true} \to 1$.

Function application requires additional translation logic, as the HIR/ANF already differentiates between thin and fat pointers. The distinction is tracked from the HIR onward because the type system already records whether a callable carries an environment; deferring it to the MIR would require re-deriving this information. Thin pointers represent calls to functions that carry no environment, notably thunks, whereas all other callables are fat pointers. A fat pointer pairs a function pointer with a captured environment, constructed when the callable is defined; because the language is referentially transparent, the captured values cannot change between construction and invocation. The MIR does not distinguish the two: reification desugars thin calls by passing arguments directly to the referenced body, and fat calls by first projecting the function pointer and environment from the closure aggregate, then prepending the environment as the first argument. Listing 3 illustrates the distinction.

HIR/ANF

```hir
// thin call (thunk, no environment)
let %0 = max:0()
in ...

// fat call (closure with captures)
let %1 = add:0(2, 3)
in ...
```

MIR

```mir
// thin: arguments passed directly
%0 = apply (max:0 as FnPtr)

// fat: project fn_ptr and env, prepend env
%1 = apply %add.0 %add.1 2 3
//          ^^^^^^^ ^^^^^^
//          fn_ptr  env
```

Listing 3: Thin and fat call desugaring during reification

The HIR represents type constructors as a dedicated node kind, as the language lacks a formal module system that could express them as ordinary definitions. A type constructor is a function that wraps a value in a nominal newtype; for unit newtypes, it takes no arguments. The MIR has no dedicated construct for this and instead lowers each type constructor into a separate body. Despite being compiler-generated and requiring no captures, the resulting body is always a closure with an empty environment, not a thin pointer as in the case of thunks. Type constructors are, from the language's perspective, ordinary closures: they can be stored in variables, passed as arguments, and participate in higher-order functions. A call site therefore cannot determine whether the callee is a thin or fat pointer and must assume a fat call convention [11:5.6].

Closures in the HIR capture variables implicitly; the MIR, by contrast, does not permit implicit capture. Reification resolves this through closure conversion, formalized in Algorithm 7. For each closure, a dependency analysis computes the set of free variables: those referenced but not defined within the closure body. Thunked bindings are excluded from this set, as they are compiler-generated module-level values with statically known locations that can be resolved directly during reification. The remaining variables constitute the capture set C. At the call site, the captured values are packed into an environment tuple and stored in a fresh local. The closure is then lowered as a separate body whose first parameter is the environment; at the callee's entry block, the environment is destructured via field projections that bind each captured variable to a fresh local. The resulting closure aggregate pairs a function pointer with the environment tuple, completing the fat pointer that call sites consume [59].

Algorithm 7: Closure conversion

procedure ConvertClosure(c, block)
  ▷ Given closure c with parameters P and body b
  $R \leftarrow \text{FreeVars}(b)$
  $D \leftarrow \text{DefinedVars}(b) \cup \text{Thunks}$
  $C \leftarrow R \setminus D$

  ▷ Pack captures into environment tuple at call site
  env ← fresh local
  env ← (captures[0], ..., captures[$|C| - 1$])

  ▷ Create body with env as first parameter
  body ← LowerBody([env] + P, b)

  ▷ Unpack environment at callee entry
  for each capture c_i in C **do**

```
    c_i := env.field(i) in entry block of body
  end

  ▷ Construct closure aggregate
  return (FnPtr(body), env)
end
```

Effectful computations follow the same pattern. The body closures of effectful operations are lowered as separate bodies with appropriate provenance, and their captures are converted through the same mechanism. Because effectful terminators reference their body closures directly rather than through function application, the environment is associated with the terminator itself rather than passed through a call site. The current implementation supports graph read operations with filter predicates; the architecture accommodates additional effect kinds without changes to the reification process.

6. Mid-Level IR

The HIR represents the programmer's intent through a tree structure that supports arbitrary sub-expressions, implicit closure captures, and nested control flow as ordinary nodes. While this is sufficient for whole-program transformation and analysis passes, such as type-checking and source-level optimization, it makes per-statement analysis infeasible: a single statement cannot be understood from its immediate context alone, and any intermediate state is easily invalidated by adjacent, seemingly unrelated, changes.

While the high-level intermediate representation in administrative normal form (HIR/ANF) addresses evaluation order and sub-expression naming (Chapter 5), the underlying tree structure persists. A conditional is still just another node beside a let-binding, with no explicit representation of branching or join points. Control flow can only be determined by interpreting node semantics, not from the structure itself. This prevents the kind of analysis the execution pipeline requires: subdividing a body at control-flow boundaries, yielding to other execution backends, potentially at join points, placing individual statements on different backends, and tracking which values are live across target transitions.

The MIR solves this by moving from a tree of nodes to a control flow graph (CFG) of basic blocks, where control flow is encoded explicitly as terminator edges. Each statement exists in a well-defined position within a block, and statements inside a basic block are always executed uninterrupted. This allows optimizations such as reordering or arbitrary subdivision of blocks — capabilities required for execution analysis and infeasible under tree-level reasoning.

A MIR program consists of a collection of bodies, each identified and referenced by an identifier unique to the program being compiled. Unlike the HIR, where closures are referenced by name or by the node itself, a body is always referenced by its identifier inside the MIR. A body encompasses more than just closures: it is also used to represent thunks, compiler intrinsics, and callables related to graph effects, such as filters. To differentiate between them, each body is assigned a source as provenance: closures, thunks, constructors, intrinsics, and graph read filters. Unlike in other compilers, this provenance guides compilation, not just debugging. For example, bodies that participate in graph effects, such as filters, undergo more aggressive inlining (Chapter 6.3.1) and are the only bodies that participate in execution analysis. Each body contains a set of basic blocks, the first of which is the implicit entrypoint, and a set of typed local

declarations. The MIR does not differentiate between locals and arguments; function arguments are simply the first locals, assigned before any code executes.

The HIR/ANF (Chapter 5) differentiates between thin and fat pointers, with fat pointers corresponding to anything that originates from a closure. Thin and fat pointers are differentiated by the data they carry: a thin pointer is just a reference to a callable, such as a thunk, whereas a fat pointer also carries an environment as its first argument. Reification erases this distinction, so the MIR treats every callable reference uniformly as a body identifier.

Implicit environment capture, still present in the HIR, is eliminated in the MIR. Inside the HIR, a closure cannot be inspected without global context such as the parent scope. This forces analyses like dead code elimination to either expensively (re-)compute environment information or conservatively assume that every binding may be used downstream in the presence of closures inside any subsequent binding. By contrast, the MIR does not permit nested closures or implicit capture. Closures are compiled to separate bodies through closure conversion: captured variables are packed into an environment tuple, passed as the first argument, and unpacked into locals at the callee's entry block. Combined with the CFG structure, closure conversion reduces analysis from global to largely local reasoning [59].

The MIR is in static single assignment (SSA) form, meaning that each local is assigned exactly once. We chose SSA for two reasons. First, analyses such as dataflow analysis are substantially simplified under SSA, as every use of a value traces to a unique definition site. Second, the referentially transparent nature of the underlying language, preserved in the HIR, is structurally close to SSA: functional programs in ANF share the single-definition property with SSA, and the correspondence between the two follows from the known equivalences between ANF, CPS, and SSA. This allows most passes to operate directly in SSA without violating it, with only some requiring an explicit repair step afterward [5, 18, 24, 39, 69].

To represent values at join points, SSA requires a mechanism for selecting definitions from different predecessors. The standard approach – formalized by Ron Cytron, et al. [18] – uses φ nodes: special statements at the start of a basic block that select a value based on which predecessor transferred control. An alternative, used in compilers such as MLIR, is block parameters: each basic block declares parameters and each incoming edge supplies arguments, with Andrew W. Appel [5] showing that the two are equivalent. We chose block parameters for the HashQL MIR because they make join-point dataflow explicit in the edge structure, rather than encoding it implicitly inside blocks. This simplifies modifications, such as adding or removing edges, and confines reasoning about block-local values to the defining block [49].

The MIR also preserves type information throughout its construction and transformation. Each local carries the type determined during HIR type checking or subsequently derived from a MIR transformation. Execution analysis and backend translation both depend on this type information. For example, the PostgreSQL database uses an encoding scheme in which certain values are ambiguous without type information (Chapter 7). Preserving types allows the compiler to determine which values can be

safely round-tripped and to inform lossless translation even in the presence of ambiguous encodings.

6.1. Instruction Set

The MIR instruction set consists of two classes of instructions: statements, which execute sequentially within a basic block and cannot influence control flow, and terminators, which end a basic block by either transferring control to a successor, returning to the caller, or yielding to the orchestrator. Similar to the HIR/ANF, the MIR does not permit nested sub-expressions; instead, instructions operate on operands, where an operand is either a place or a constant [26, 56].

A constant is a value known at compile time that can be embedded directly into an instruction without requiring any intermediate construction or indirect access. The MIR distinguishes four constant kinds: integers, primitives, unit values, and function pointers. Function pointers are created during reification and reference other bodies. Unlike all other constants, they have no direct equivalent in the HIR; they correspond instead to body sources such as thunks or closure definitions. Unit values correspond to empty tuples and the `Null` value. This semantic collapse of equivalent empty-value representations addresses a flaw in the initial language design, Chapter 10.1.3.2 discusses a planned resolution at the type-system level [11].

The most consequential distinction is between integers, booleans, and primitives: HashQL, like Python, supports arbitrary-precision integers, but optimization passes require a bounded representation that is efficient to manipulate, compare, and store. The MIR therefore introduces a dedicated signed 128-bit integer constant. Any value that exceeds this range is, like any other primitive, relegated to an opaque primitive constant – no optimization pass inspects or transforms these values. Boolean values are also lowered into integer constants, but retain a tag for type preservation. The compiler can therefore treat every compile-time value uniformly, with boolean-integer coherence already guaranteed by HIR type checking [11].

A place denotes a path to a storage location within a body. The concept is modeled after the Rust MIR, where a place consists of a local, which identifies the root variable, followed by zero or more projections that navigate into nested data. Each projection carries its own result type, so the type at any point along the path is statically known. Any subsequent operation, such as an optimization pass, can therefore resolve a subcomponent reference without re-deriving type information from context [56].

The MIR supports three kinds of projections, two of which follow the Rust MIR design. A field projection accesses a positional element within a tuple or within a closed struct (Chapter 10.1.4.2). An index projection navigates into a dynamic container such as a list or dictionary, where the index must be supplied as a local, to enforce that the index value is computed before the projection is accessed. This work adds a third kind: a name projection, which accesses a field by symbol rather than by position and is used for structural types where only the field name is guaranteed.

```mir
thunk {thunk@0}() -> Integer {
    bb0(): {
        return 42
    }
}

fn {closure@1}(%0: (Integer), %1: Integer) -> Foo {
    let %2: Integer
    let %3: Boolean
    let %4: Integer
    let %5: Integer
    let %6: Foo

    bb0(): {
        %2 = %1 + %0.0
        %3 = %2 >= 2

        switchInt(%3) -> [0: bb2(), 1: bb1()]
    }

    bb1(): {
        goto -> bb3(%1)
    }

    bb2(): {
        %4 = apply ({def@0} as FnPtr)

        goto -> bb3(%4)
    }

    bb3(%5): {
        %6 = opaque(Foo, %5)

        return %6
    }
}
```

Listing 4: MIR program illustrating the instruction set

6.1.1. Statement

The MIR has two statement kinds: assignment and nop. An assignment evaluates an rvalue – an expression that produces a value without allocating named storage – and writes the result into a destination place. Each rvalue operates on one or more operands. We define six rvalue kinds: loads, binary operations, unary operations, aggregate construction, input access, and function application.

The destination of an assignment is not restricted to a bare local and may also target a projection, such as a field within a tuple. During compilation, however, the MIR operates in strict SSA: each local is defined exactly once, and the current pipeline does not emit projection assignments. In anticipation of SSA breaking operations, the dataflow framework classifies such writes as partial definitions, so that post-SSA phases can perform in-place sub-component updates without reconstructing entire aggregates. Implementing this capability is deferred to the register-based bytecode VM discussed in Chapter 10.1.1.2.

The nop statement performs no computation. Transformation passes, such as dead store elimination (DSE), replace dead statements with nops rather than removing them, because statement indices are used as location references throughout the pipeline; in-place replacement avoids invalidating those references. Subsequent cleanup passes remove these nops when necessary.

Load. A load is the simplest rvalue: it reads an operand and writes it directly into the destination place. In the case of a place, the value at the specified location is copied; in the case of a constant, the value is materialized. Loads do not perform any computation and instead serve as the primary mechanism for transferring data.

```mir
%3 = %1.0[%2].foo
%4 = 1
```

Listing 5: Load from a projection chain and from a constant

Binary Operation. A binary operation applies an operator to two operands and produces a single result. The operator set covers arithmetic (+, −), comparison (=, ≠, <, ≤, >, ≥), and bitwise operations (&, |). Additional operators, such as exponentiation, are defined in the representation but not implemented, because the HIR does not expose them. The Großer Beleg limited the initial operator set to reduce implementation complexity at the cost of expressiveness [11].

Unlike the HIR, the MIR has no need to define logical (∧, ∨) operators. Logical operations require short-circuit evaluation: given $A \wedge B$, B must only be evaluated if A is true. To reduce the operator set, during HIR normalization (Chapter 5) logical expressions with nested sub-expressions are lowered into explicit conditionals. The only surviving logical operations therefore operate on ANF atoms, which are by definition free of nesting. Because booleans are represented as integers of bit-width 1 inside the MIR, bitwise operations over these integers are equivalent to their logical counterparts, as shown in Table 2, making dedicated logical operators superfluous.

```mir
%2 = %0 + %1
%3 = %2 >= 2
```

Listing 6: Binary addition and comparison

A	B	$A \wedge B$	$A \,\&\, B$	$A \vee B$	$A \mid B$
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1

Table 2: Logical and bitwise truth table

Unary Operation. A unary operation applies an operator to a single operand. The available operators are bitwise not and arithmetic negation. By the same reasoning as logical conjunction and disjunction, logical negation is not a separate operator: for a single-bit integer, bitwise not flips $0 \to 1$ and $1 \to 0$, which is identical to logical not.

```mir
%1 = ~%0
%2 = -%0
```

Listing 7: Bitwise not and arithmetic negation

Aggregate Construction. An aggregate constructs a composite value from its components. The MIR supports six aggregates: tuples, structs, lists, dicts, opaques, and closures. Tuples are fixed-size heterogeneous containers; structs extend them by associating each position with a field name. A list is a dynamically sized but homogeneous counterpart of a tuple; a dictionary is a key-value collection that takes its operands in alternating key-value pairs. The remaining two aggregates are more specialized: an opaque wrapper places a value under a nominal type boundary, corresponding to a newtype in the type system, while a closure constructs a fat pointer from exactly two operands – a function pointer and the captured environment.

```mir
%2 = (%0, %1)
%4 = opaque(::core::uuid::Uuid, %3)
%7 = closure(%5, %6)
```

Listing 8: Tuple, opaque wrapper, and closure construction

Input. An input is a form of dependency injection that allows the host to supply named, typed values at the start of execution. Inputs are a query language convenience: rather than threading configuration through parameter structs, the program accesses externally supplied values by name, analogous to process environment variables on UNIX systems. Two operations are available: load retrieves the value, while exists returns a boolean indicating whether a value has been provided [11].

```mir
%0 = input LOAD foo
%1 = input EXISTS foo
```

Listing 9: Loading and checking an input parameter

Function Application. A function application invokes a callable with a list of arguments. Its callable is a regular operand that must resolve to a function pointer. At the callsite, no differentiation is made between thin and fat pointers; all function applications assume thin pointers. Reification, which erases that differentiation, desugars closure calls so that the environment is prepended as the first argument.

HashQL is referentially transparent and does not support unwinding or exceptions. Function application therefore has a single continuation and no alternate control-flow paths; terminator status is unnecessary. By contrast, the Rust MIR models calls as terminators because they may unwind, abort, or otherwise diverge [56].

```mir
%1 = apply ({def@0} as FnPtr) %0
%3 = apply %2 %0 %1
```

Listing 10: Direct and indirect function application

6.1.2. Terminator

A terminator ends a basic block by directing control flow. The HashQL MIR defines four control-flow terminators, which are distinct from effectful terminators in that they neither introduce external data nor suspend execution. Each terminator specifies one or more targets, where a target specifies the destination basic block and the arguments to be mapped to that block's parameters.

Goto. A goto is an unconditional jump to a single target.

```mir
goto -> bb0(%1, 2, %0.1)
```

Listing 11: Unconditional jump with block arguments

SwitchInt. A switch-integer terminator evaluates a discriminant and selects one of several targets based on the discriminant's value. Each target is associated with an integer constant, with an optional otherwise target for values not explicitly listed. We chose a switch-integer terminator over a dedicated if-else because it generalizes to arbitrary multi-way branching, and future pattern matching in the HIR (Chapter 10.1.3.1) will be able to lower without introducing a new terminator. Because booleans are lowered to 1-bit integers, as described above, conditional branches are handled uniformly through the same mechanism.

```mir
switchInt(%0) -> [0: bb1(), 1: bb2(2, 0), _: bb3(%0)]
```

Listing 12: Multi-way branch with an otherwise target

Return. A return terminates the current body and passes a value back to the caller.

```mir
return %0
```

Listing 13: Return with a value

Unreachable. An unreachable terminator marks a block as unreachable. Entering such a block is considered undefined behavior (UB). Unreachable terminators are not generated during reification but are introduced by optimization passes as placeholders for eliminated branches. As with nop statements, this defers block removal to a subsequent cleanup pass, avoiding the need to update block indices immediately.

```mir
unreachable
```

Listing 14: Unreachable block marker

6.1.3. Effectful Terminator

Effectful terminators, unlike their control-flow counterparts, suspend execution and hand control to the orchestrator. The mechanism is similar to the yield terminator in the Rust MIR, but where Rust's yield returns control to the caller, an effectful terminator suspends the entire call stack and yields to the orchestrator directly. The orchestrator then fulfills the request and resumes execution at the successor block. This is also what distinguishes effectful terminators from function application: an apply evaluates within the MIR's execution model without crossing a control-flow boundary, whereas an effect exits that model entirely and re-enters through the orchestrator [56, 64].

These terminators do not carry fully materialized targets. Instead, they specify only a successor block, and the orchestrator injects the result into that block's parameters. The number and layout of injected parameters depend on the specific effect. Because effectful terminators do not participate in the standard target mechanism, the small subset of passes that inspect target edges – most notably SSA repair and CFG simplification – must handle them as opaque boundaries. Most passes require no special treatment, as they never inspect CFG edges directly.

GraphRead. The graph read terminator is currently the only effectful terminator and follows a head-body-tail design implementing a map-reduce-like pattern. The head establishes the data source and binds a temporal axis. While support for multiple data sources is planned, only entity data sources are currently supported. Body operators sequentially transform the result set by operating on each entity individually rather than on the complete set; this is the map phase, with a filtering predicate implemented as part of this work. The tail materializes the result, corresponding to the reduce phase; this work implements the collect operator. Once evaluated, the value produced by the tail is passed as the first argument to the successor block. The pipeline has been designed to accommodate additional operators in the future (Chapter 10.1.2.1) [11:5.6.1.3].

Filter predicates are compiled as separate bodies. Each body carries its origin as provenance, which passes use to specialize their behavior – inlining, for example, applies more aggressively to filter bodies than to general closures. Because filters are standalone bodies rather than expressions nested inside the graph read, transformation passes can optimize and analyze them without any special-casing and independent of the terminator that references them. This separation also enables future optimizations, such as consolidating duplicate predicates across terminators, which are not yet implemented.

```mir
graph read entities(%0)
    |> filter({graph::read::filter@0}, %1)
    |> collect -> bb1(_)
```

Listing 15: Graph read with a filter predicate

6.2. Dataflow Analysis

The MIR relies on dataflow analysis to drive both optimization passes and execution analysis. The framework implementing dataflow has been adapted from the Rust compiler's implementation and provides a generic worklist-based fixpoint iteration over a lattice-parameterized domain, supporting both forward and backward analyses. Four structural adaptations distinguish the framework implemented as part of this work from its origin: transfer functions for block parameters, which model argument-to-parameter binding at CFG edges; transfer functions for effectful terminators; an externalized lattice, where the algebraic structure is separate from the carrier type so that the same domain can support multiple interpretations; and optional iteration metadata, which allows individual analyses to bound fixpoint iteration. The last adaptation addresses analyses where unbounded block-level fixpoint iteration would delay convergence without proportional precision gains, such as size estimation (Chapter 7.3). Bounded iteration reduces the precision of the resulting approximation but preserves a safe lower bound: the estimate is conservative with respect to the cost model that consumes it. The output of each analysis run is a pair of entry and exit states per basic block over the chosen domain [41, 70].

Liveness Analysis. The framework is used to implement standard backward liveness analysis using a gen/kill transfer function. The analysis computes which locals are live at each block boundary [3:9.2.5].

Traversal-Aware Liveness Analysis. Traversal-aware liveness analysis extends the standard analysis by tracking two independent dimensions in parallel. This analysis addresses a granularity mismatch: standard liveness determines which locals are live at each program point, but execution analysis additionally requires knowing which fragments of data a graph operator body accesses. Individual graph operators retrieve data by projecting through the vertex function parameter, and because of the heterogeneous nature of the system, each field may reside on a different backend; transferring the full vertex at every boundary would incur unnecessary transfer and computation cost. An analysis that feeds execution analysis must therefore track which paths are actually used, so that only the necessary data is fetched. To achieve this, the analysis operates over a parallel domain consisting of a local liveness bitset and a per-vertex-path bitset. The vertex local is excluded from the local bitset to avoid double-counting: without this exclusion, the vertex would be counted once as a live local and once through its individual paths. To account for cases where the vertex is required to be fully loaded, such as the vertex local appearing without a projection, or a projection not mapping to a pre-defined storage location, all paths are conservatively marked live. This preserves path-based access on vertices while maintaining the local-based liveness analysis for all other locals.

```mir
fn {filter@0}(%0: (), %1: Entity) -> Bool {
  let %2: Bool
  let %3: ?
  let %4: Bool

  bb0():                          // ▷ {} {Properties, Archived}
  {
    %2 = load true                // kill %2
    switchInt(%2) -> [            // gen  %2
      0: bb2(), 1: bb1()
    ]
  }                               // ◁ {} {Properties, Archived}

  bb1():                          // ▷ {} {Properties}
  {
    %3 = %1.properties            // gen path Properties, kill %3
    %4 = %3 != null               // gen %3, kill %4
    goto -> bb3(%4)               // gen %4
  }                               // ◁ {} {}

  bb2():                          // ▷ {} {Archived}
  {
    %4 = %1.metadata.archived     // gen path Archived, kill %4
    goto -> bb3(%4)               // gen %4
  }                               // ◁ {} {}

  bb3(%5):                        // ▷ {} {}
  {                               // kill %5 (block param)
    return %5                     // gen %5
  }                               // ◁ {} {}
}
```

Listing 16: Traversal-aware liveness on a diamond CFG

The two dimensions are visible in the Listing 16: the vertex local `%1` never appears in the local set despite being accessed in both `bb1` and `bb2`; its field accesses are tracked separately as entity paths. At the branch point `bb0`, paths from both successors are joined, producing the union `{Properties, Archived}`. Entry and exit states, marked ▷ and ◁, illustrate that path-level tracking propagates correctly through join points.

The consequence is partial entity hydration. An entity carries properties, temporal metadata, provenance records, and type identifiers, among others. By tracking the paths accessed, graph operator bodies load only the data they require, eliding serialization and transfer of unused fields. Rather than treating the vertex as a monolithic value, the cost model and backend translation can operate at field-level granularity. The analysis is not a general-purpose access-path abstraction: the path lattice is fixed to the entity schema rather than derived from the program's type structure, and the paths carry backend-origin semantics that feed directly into transfer cost estimation.

This domain-specific restriction is what makes the analysis tractable and useful for placement; a fully general access-path-sensitive analysis would introduce unnecessary complexity for the fixed set of entity projections this system supports. The mechanism by which these path sets feed into execution analysis is described in Chapter 7.

Execution Analysis. Execution analysis extends the dataflow framework beyond liveness: size estimation and supported-statement analysis both use forward analyses over custom domains. Size estimation uses a forward analysis with affine equations over function parameters to track value sizes through the program, in cases where a type's size cannot be statically determined (Chapter 7.3). Supported statement analysis uses forward analysis to determine if a local is supported, depending on the operands used in a statement (Chapter 7.2). The dataflow framework presented here provides the shared infrastructure for both.

6.3. Transformation Pipeline

The CFG-based SSA design of the MIR admits standard optimization techniques. Rather than implementing all of those present in a typical compiler, this work selects the subset that contributes most to inlining and groups them under a canonicalization pass. The resulting transformation pipeline is structured around inlining itself (see Figure 3).

Figure 3: Transformation Pipeline

The transformation pipeline prepares the MIR body for execution analysis, which assigns basic blocks to backends, operating at instruction granularity within individual bodies. Within execution analysis, the central obstacle is function application (Chapter 6.1.1.6). Languages with divergent control flow, such as Rust, model function calls as terminators with distinct edges for normal and exceptional outcomes; languages without divergence, including HashQL, model them as statements; some, such as Swift through the SIL, implement both. Neither choice eliminates the difficulty: both force execution analysis to reason across body boundaries, differing only in granularity [26, 56].

If function application were a terminator, execution analysis would need to track each invocation as a triple of $(\text{caller}, \text{callee}, \text{operands})$. Because the operands are local to the caller, the analysis cannot memoize across call sites; a deep call chain produces a distinct analysis state at every level. During placement, the solver would need to traverse into each callee and resume at the caller, performing inlining in all but name during both analysis and evaluation. If function application remains a statement, the entire callee body collapses into a single placement decision: the callee must execute on one backend, and its cost is charged as a lump sum. This inflates the weight of call sites relative to their surrounding scope and produces a coarser placement than necessary, because the callee's internal operations are confined to a single backend. In both cases, the analysis can only proceed when the callee is statically known. The result

is equivalent to inlining in both: at full granularity for terminators, at body granularity for statements. The downside of either approach is an explosion in analysis state and correctness surface, both caused by operating on a pseudo-inlined body. Alternative designs, such as callee summaries or context-sensitive interprocedural placement, would avoid inlining but require the solver to reason about operand-specific transfer costs and backend-dependent support constraints across call boundaries, complexity that scales with call depth and context sensitivity. This work instead inlines aggressively before execution analysis begins, making the problem intra-procedural by construction and confining all placement decisions to a single body.

6.3.1. Inlining

Inlining serves two purposes in this work. As established above, it resolves the inter-procedural problem: by substituting function calls with the callee's body, execution analysis gains visibility into operations that would otherwise be opaque, and can assign them to backends individually. The second purpose is traditional: for small functions, call overhead – especially under a tree-walking interpreter such as the one in this work – exceeds the cost of duplicating the body, and inlining eliminates that overhead while exposing further optimization opportunities to subsequent canonicalization passes [3:12].

Because these two goals operate on the same mechanism but differ in their trade-offs, the inlining pass described here applies two discrete phases. The normal phase uses heuristic scoring and a per-caller budget to inline where the benefit outweighs the code size increase. The aggressive phase forgoes heuristics and transitively inlines all eligible calls within graph operator bodies, because the more operations exposed to execution analysis, the more work can be dispatched to remote backends rather than falling back to the interpreter.

To guide inlining decisions, the pass operates over a directed call graph G whose vertices are the program's bodies and whose edges point from caller to callee. Each edge carries a location and a kind: Apply for direct function application, Filter for graph operator filter invocations, and Opaque for any other body reference, including function-pointer constants and incidental references.

$$
\begin{aligned}
G &= (V, E) \\
E &\subseteq V \times V \times L \times K \\
K &= \{\text{Apply}, \text{Filter}, \text{Opaque}\} \\
D &= \{\text{Always}, \text{Heuristic}, \text{Never}\}
\end{aligned}
$$

$$
\begin{aligned}
\text{blocks}(b) &= \text{basic blocks of } b \\
\text{rvalues}(k) &= \text{rvalues of statements in } k \\
\text{term}(k) &= \text{terminator of } k \\
\text{targets}(t) &= \text{targets of SwitchInt terminator } t
\end{aligned}
\tag{2}
$$

$$
\begin{aligned}
\text{callers}(b) &= \{(s, l) \mid (s, b, l, \text{Apply}) \in E\} \\
\text{callees}(b) &= \{(f, l) \mid (b, f, l, \text{Apply}) \in E\} \\
\text{is\_unique}(b) &= [\,|\,\text{callers}(b)| = 1\,] \\
\text{is\_leaf}(b) &= [\forall (f, l) \in \text{callees}(b) : f \text{ is intrinsic}] \\
\text{directive}(b) &\in D
\end{aligned}
$$

Figure 4: Call graph and utility definitions

Normal Inlining. The normal phase uses a heuristic-based, budget-limited algorithm, where each body is processed exactly once: callees are inlined before their callers, so that when a caller is visited its callees have already been processed. To avoid the need for transitive inlining and reduce redundant work, each body tracks its approximate size as an accumulated cost, updated after each inlining so that subsequent callers see the true post-inlining size. The initial size of each uninlined body is estimated by summing weighted contributions for each construct, where w_x denotes the constant weight for construct x. The cost accounts for statements, terminators, and the total number of blocks: control-flow transitions between blocks carry overhead, and a highly fragmented body is less likely to benefit from inlining than one with little control flow.

$$
w_{\text{term}}(k) = \begin{cases}
w_{\text{t\_switch\_base}} + w_{\text{t\_switch\_branch}} * |\,\text{targets}(\text{term}(k))| & \text{if term}(k) = \text{SwitchInt} \\
w_{\text{t\_graphread}} & \text{if term}(k) = \text{GraphRead} \\
w_{\text{t\_goto}} & \text{if term}(k) = \text{Goto} \\
w_{\text{t\_return}} & \text{if term}(k) = \text{Return} \\
w_{\text{t\_unreachable}} & \text{if term}(k) = \text{Unreachable}
\end{cases}
$$

$$
w_{\text{rvalue}}(r) = \begin{cases}
w_{\text{rv\_load}} & \text{if } r = \text{Load} \\
w_{\text{rv\_binary}} & \text{if } r = \text{Binary} \\
w_{\text{rv\_unary}} & \text{if } r = \text{Unary} \\
w_{\text{rv\_aggregate}} & \text{if } r = \text{Aggregate} \\
w_{\text{rv\_input}} & \text{if } r = \text{Input} \\
w_{\text{rv\_apply}} & \text{if } r = \text{Apply}
\end{cases}
\tag{3}
$$

$$
\text{size}(b) = \sum_{k \in \text{blocks}(b)} \left(w_{\text{block}} + w_{\text{term}}(k) + \sum_{r \in \text{rvalues}(k)} w_{\text{rvalue}}(r) \right)
$$

Figure 5: Body cost estimation

This cost is updated after each inlining with the caller's cost increasing by the callee's cost minus the removed `Apply` cost, so subsequent inlining decisions see the accumu-

lated size. The estimate is a deliberate under-approximation: the block split introduced by inlining (an additional terminator and block), parameter-binding loads, possible result write-back temporaries, and the rewriting of `return` terminators to `goto` (which may differ in weight) are not accounted for. In practice, the resulting bias toward acceptance is marginal and consistent with the pipeline's preference for maximizing inlined scope before execution analysis.

To ensure that callees are processed before their callers, the inliner computes the condensation of G and traverses the resulting directed acyclic graph (DAG) in reverse topological order. The fundamental issue is that recursive calls produce non-trivial strongly connected components (SCCs) in the condensation, within which, by definition, no topological order exists: every member is reachable from every other, and naively inlining all intra-SCC calls would cause the inliner to diverge at compile time. The conservative approach is to never inline within SCCs, but this severely limits optimization of mutually recursive functions. While some compilers address this through bounded recursive inlining controlled by depth and size thresholds, this work instead adopts a technique inspired by GHC's occurrence analysis to break these cycles and establish a dependency order within each SCC [62, 78].

For non-trivial SCCs (two or more members), where mutual recursion prevents a topological order, the inliner selects a set of loop breakers. A loop breaker is a body to which calls within the SCC are not inlined, with the breaker still being able to inline calls to non-breakers. We choose the loop breakers greedily, as finding a minimum feedback vertex set is NP-hard, and the greedy approach is tractable for the small SCCs that arise in practice [38]. Each body is scored by an approximation of its inverse inlining value:

$$\begin{aligned} \text{breaker}(b) = {} & w_{\text{b\_size}} * \text{size}(b) \\ & - w_{\text{b\_caller}} * |\,\text{callers}(b)| \\ & - w_{\text{b\_unique}} * \text{is\_unique}(b) \\ & - w_{\text{b\_leaf}} * \text{is\_leaf}(b) \\ & + \begin{cases} +\infty & \text{if directive}(b) = \text{Never} \\ -\infty & \text{if directive}(b) = \text{Always} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

Figure 6: Loop breaker scoring

where $w_{\text{b\_size}}$ is the size weight, $w_{\text{b\_caller}}$ the caller penalty, $w_{\text{b\_unique}}$ the unique-callsite penalty, and $w_{\text{b\_leaf}}$ the leaf penalty. Members are greedily selected as breakers in descending score order until the remaining subgraph forms a DAG, which is then reverse-topologically ordered, with breakers appended last. Because breakers can only inline non-breakers intra-SCC, processing them last ensures that all their inlineable callees have already been optimized.

Once an inlining order has been chosen, each body is processed in order, with self-calls never being eligible for inlining, as the inliner does not perform tail-call optimization (Chapter 10.1.4.5). Each body carries an inline directive derived from its source provenance: Always for constructors, Never for intrinsics and graph operator bodies, and Heuristic for closures and thunks. The following score function accounts for directives, cost thresholds, and heuristic bonuses:

$$
\begin{aligned}
\text{let } c &= (s, f, l, \text{Apply}) \in E \\
\text{single\_caller}(c) &= [\{s' \mid (s', l') \in \text{callers}(f)\} = \{s\}] \\
\text{in\_loop}(c) &= [l \text{ is inside a loop body}] \\
\text{score}(c) &= w_{\text{s\_loop}} * \text{in\_loop}(c) \\
&\quad + w_{\text{s\_leaf}} * \text{is\_leaf}(f) \\
&\quad + w_{\text{s\_single}} * \text{single\_caller}(c) \\
&\quad + w_{\text{s\_unique}} * \text{is\_unique}(f) \\
&\quad - w_{\text{s\_size}} * \text{size}(f) \\
&\quad + \begin{cases} +\infty \text{ if directive}(f) = \text{Always} \vee \text{size}(f) < w_{\text{s\_trivial}} \\ -\infty \text{ if directive}(f) = \text{Never} \vee \text{size}(f) > w_{\text{s\_max}} * \begin{cases} w_{\text{s\_loop\_mult}} & \text{if in\_loop}(c) \\ 1 & \text{otherwise} \end{cases} \\ 0 \quad \text{otherwise} \end{cases}
\end{aligned}
\tag{5}
$$

Figure 7: Callsite scoring

where c is an Apply edge with caller s, callee f, and location l; $w_{\text{s\_size}}$ is the size penalty factor; and each $w_{\text{s\_}}$ term is the weight for the corresponding property. Callsites with negative scores are rejected. Those with infinite scores ($+\infty$) are inlined unconditionally, bypassing the budget. The remaining callsites (finite non-negative scores) are ranked in descending order and selected while the callee's cost fits within the per-caller budget ($w_{\text{s\_max}} * w_{\text{s\_budget}}$). Each inlined callee consumes its body cost from the budget, which prevents code-size explosion from a sequence of individually profitable inlinings.

Aggressive Inlining. Graph operator bodies participate in the normal inlining order described above, but bypass heuristics entirely: every eligible callsite is inlined regardless of score or budget. Once the normal phase completes, a second iterative phase begins for graph operator bodies specifically. In each iteration, the body is traversed for eligible callsites, where a callsite is eligible if the callee is statically known and is not a loop breaker in any SCC; the loop-breaker constraint is necessary because the iterative phase may inline across multiple SCCs in the condensation, and without it the process would diverge on recursive call chains. Iteration continues until no eligible callsites remain or a configurable upper limit is reached. While convergence is guaranteed, the upper limit bounds code growth.

Inlining Procedure. The block-parameter-based SSA form makes the inlining transformation straightforward. The procedure uses the following utility definitions in addition to those in Figure 4:

$$
\begin{aligned}
\mathrm{bb}_i^A &\coloneqq i\text{-th basic block of body } A \\
\mathrm{block}(l) &\coloneqq \text{basic block containing location } l \\
\mathrm{statements}(b) &\coloneqq \text{statements of block } b \\
&= \langle s_0, \ldots, s_n \rangle \\
\mathrm{params}(b) &\coloneqq \text{parameters of block } b \\
&= \langle p_0, \ldots, p_n \rangle \\
\mathrm{locals}(A) &\coloneqq \text{locals of body } A \\
&= \langle \%_0, \ldots, \%_n \rangle
\end{aligned}
\tag{6}
$$

Figure 8: Inlining utility definitions

Given a caller A inlining callee B at callsite $c = (A, B, l, \mathrm{Apply}) \in E$, we define $n = |\mathrm{blocks}(A)|$ and $m = |\mathrm{blocks}(B)|$ before any modification. Let j denote the zero-based index of the callsite statement within $\mathrm{block}(l)$. The statement at j has the form $\%_r = \mathrm{Apply}(f, a_0, \ldots, a_{p-1})$ where p is the number of arguments, $\%_r$ is the result local, and a_i are the argument operands. The transformation proceeds in five steps:

1. **Split.** Let $S = \mathrm{statements}(\mathrm{block}(l))$, with $S_{\mathrm{before}} = \langle S_0, \ldots, S_{j-1} \rangle$ and $S_{\mathrm{after}} = \langle S_{j+1}, \ldots, S_{|S|-1} \rangle$. Create a continuation block bb_n^A with $\mathrm{params}(\mathrm{bb}_n^A) = \langle \%_r \rangle$, $\mathrm{statements}(\mathrm{bb}_n^A) = S_{\mathrm{after}}$, and $\mathrm{term}(\mathrm{bb}_n^A) = \mathrm{term}(\mathrm{block}(l))$. If the left-hand side involves projections (e.g. $\%_r.f$), a fresh local $\%_t$ replaces $\%_r$ as the continuation block parameter, and $\%_r.f = \mathrm{load}\ \%_t$ is prepended to S_{after}. Set $\mathrm{statements}(\mathrm{block}(l)) = S_{\mathrm{before}}$ and $\mathrm{term}(\mathrm{block}(l)) = \mathrm{goto} \to \mathrm{bb}_{n+1}^A()$.
2. **Bind.** Let $\lambda = |\mathrm{locals}(A)|$. For each argument a_i with $0 \le i < p$, append $\%_{\lambda+i} = \mathrm{load}\ a_i$ to $\mathrm{block}(l)$.
3. **Copy.** Append B's basic blocks to A as $\mathrm{bb}_{n+1}^A, \ldots, \mathrm{bb}_{n+m}^A$ and B's local declarations to A's.
4. **Rename.** For all references in $\mathrm{bb}_{n+1}^A, \ldots, \mathrm{bb}_{n+m}^A$: offset each local $\%_k$ to $\%_{k+\lambda}$ and each block bb_k^B to bb_{k+n+1}^A.
5. **Redirect.** For each $\mathrm{bb}_i^A \in \{\mathrm{bb}_{n+1}^A, \ldots, \mathrm{bb}_{n+m}^A\}$ where $\mathrm{term}(\mathrm{bb}_i^A) = \mathrm{return}\ o$, replace with $\mathrm{term}(\mathrm{bb}_i^A) = \mathrm{goto} \to \mathrm{bb}_n^A(o)$.

6.3.2. Canonicalization

Canonicalization is a composition of standard compiler optimization passes, covering simplification, value propagation, and dead code elimination, applied globally across all bodies in a program. The label "canonical" therefore denotes irreducibility under these transformations, not conformance to an externally specified normal form.

The pipeline applies canonicalization on both sides of inlining, for asymmetric reasons. Before inlining (Figure 3), canonicalization prepares the program for the inliner: simplified bodies present more accurate size information to the cost model, while propagation and branch elimination expose additional callsites. After inlining, canonicalization prepares the program for execution analysis: it eliminates redundancy that inlining introduces and exploits optimization opportunities that inlining surfaces; the result is bodies with fewer instructions and simpler control flow. Because execution analysis splits at non-consecutive backend assignments, fewer instructions and simpler control flow directly reduce fragmentation and yield fewer islands and better placement

decisions. In both cases, canonicalization is the sole content of its respective pipeline stage, which reflects the scope of this work rather than a design constraint; additional passes such as global value numbering (GVN) could be incorporated into either stage.

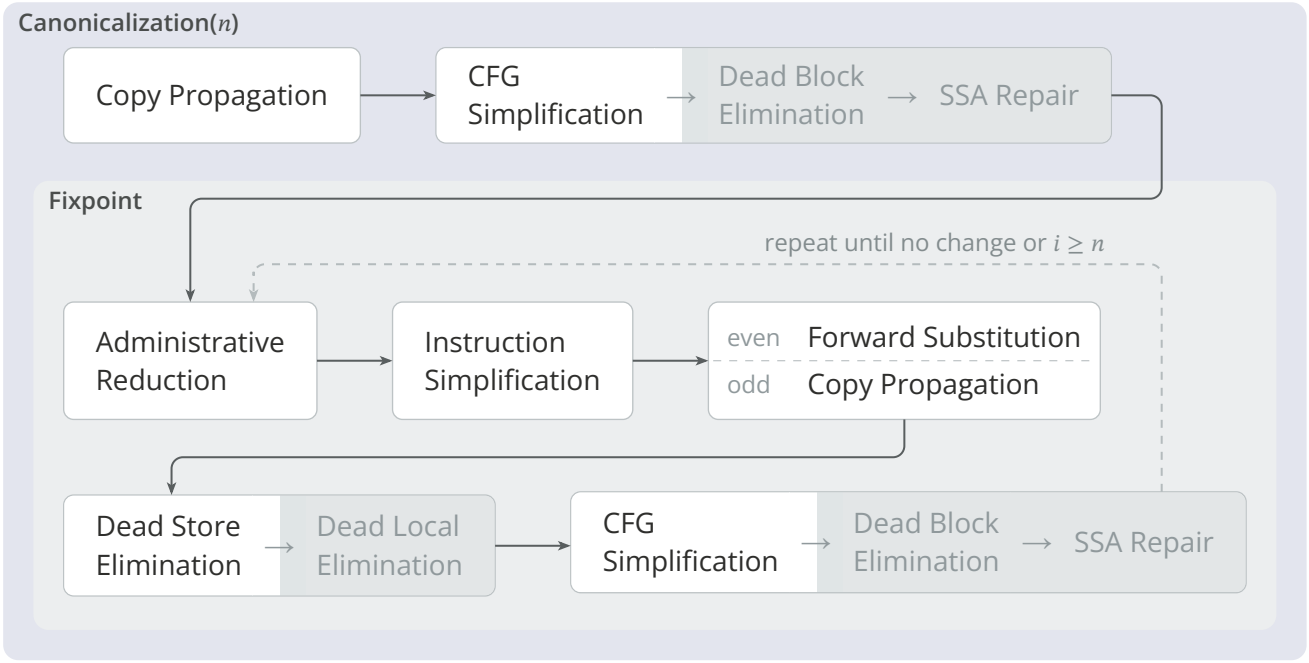


Figure 9: Canonicalization

Figure 9 shows the internal structure of the transformation pass, which proceeds in two phases. A bootstrap phase runs copy propagation followed by CFG simplification. Both are inexpensive and effective on obvious redundancy: copy propagation exposes constant conditions that CFG simplification benefits from when pruning unreachable branches and merging straight-line blocks. This shrinks bodies before the more expensive passes in the fixpoint loop operate on them. The loop itself iterates the full pass sequence up to n times, where $n = 8$ before inlining and $n = 16$ after. The limit is a worst-case bound, not the expected iteration count; in practice, most bodies stabilize within one to three rounds. The higher post-inline limit reflects that inlining surfaces nested opportunities that may require additional rounds to resolve.

The pass ordering within each iteration is chosen so that each pass feeds the next. Administrative reduction (AR) removes indirection through bodies and normalizes their shape. Instruction simplification (IS) folds constants and applies algebraic identities. Value propagation follows, alternating between forward substitution (FS) on even iterations and copy propagation (CP) on odd: forward substitution resolves deeper dependency chains through projections but is more expensive to run, and appears on even iterations because the bootstrap phase implicitly provides the initial copy propagation run. This alternation balances accuracy against speed. Dead store elimination (DSE) then removes stores whose targets are no longer live after propagation, and CFG simplification merges blocks and prunes unreachable edges exposed by the preceding passes. A different ordering would not affect correctness, but would require additional iterations to surface the same opportunities.

The fixpoint is guaranteed to converge. Most passes are strictly monotone: they simplify or remove constructs but never introduce new complexity, and the MIR is finite. The

sole exception is SSA repair, which may introduce new locals and block parameters to restore the single-definition property after CFG simplification; because it is idempotent on valid SSA, it stabilizes immediately and does not impede convergence. Because each pass simplifies along a dimension no other pass reverses, the composition is also monotone; the convergence properties of each pass follow from the analysis in its respective section.

Administrative Reduction (AR). Administrative reduction (AR) is a form of partial evaluation that eliminates structural indirection introduced by earlier compilation phases. The term originates from the lambda calculus, where an administrative redex is a β-redex, the application of a lambda to an argument, created by a program transformation rather than by the programmer. An analogous situation arises in this pipeline: the HIR/ANF normalization (Chapter 5) introduces wrapper bodies, thunks for module globals and closures that forward to another function, as structural artifacts of the translation. To enable future parallelisation and reduce per-pass complexity, most analyses in the canonicalization loop are body-local rather than program-global. Consequently, these wrappers create opaque boundaries that propagate through the pipeline: canonicalization cannot simplify across them, reducing the effectiveness of inlining and limiting the visibility of execution analysis. Work that could be dispatched to a remote backend spills to the interpreter instead.

The pass that benefits most from removing these wrappers is inlining. Thunks for global module variables delay execution, but typically return a closure pointer for a global closure that is referenced elsewhere. The inliner cannot see that the function application targets a statically known function, because the thunk boundary makes the callee appear opaque. As a result, callsites that would otherwise qualify for inlining are rejected. Running inlining inside the canonicalization loop would resolve this, but requires cross-fixpoint tracking between a program-global pass and the body-local loop, which is not feasible. administrative reduction instead runs as a dedicated pre-pass before canonicalization: it resolves the indirection once, so that the inliner and all subsequent body-local passes operate on the exposed definitions.

To enable this transformation, administrative reduction is designed as a global pass that classifies each body into one of two reducible forms and inlines reducible bodies at their call sites. Unlike the heuristic-driven inliner (Chapter 6.3.1), administrative reduction operates outside any cost equation: the bodies it targets are unconditionally beneficial to inline because they contribute no computation beyond forwarding. The mechanical transformation is analogous to the inlining procedure described in Chapter 6.3.1, but specialized for single-block bodies: the callee's statements are spliced directly into the caller at the call site, and the call is replaced with a load of the callee's return value. No new blocks or control-flow edges are introduced.

Algorithm 8: Administrative reduction classification

```
1: function Classify(b)
2:     if b has more than one basic block then
3:         return ⊥
4:     end
5:
```

```
    s ← statements of the entry block of b
    t ← terminator of the entry block of b

    ▷ Trivial thunk: all statements are loads or aggregates, returns immediately
    if all s_i ∈ s are Load or Aggregate, and t is Return then
        return TrivialThunk
    end

    ▷ Forwarding closure: trivial prelude followed by a single call
    if all s_i ∈ s except the last are Load or Aggregate then
        if last s_i is Apply, and t returns the result of that Apply then
            return ForwardingClosure
        end
    end

    return ⊥
end
```

Algorithm 8 formalizes the classification into two reducible forms:

1. **Trivial thunk.** A single basic block containing only loads and aggregate constructions, with no calls or arithmetic, that returns immediately.
2. **Forwarding closure.** A single basic block with a trivial prelude of loads and aggregates, followed by a single function application whose result is immediately returned.

Both forms are artifacts of the HIR/ANF thunk and closure construction (Chapter 5.5), which this pass reverses.

Before

```mir
fn {closure@0}(%0: Integer) -> Integer {
    bb0(): {
        return %0
    }
}

fn {closure@1}() -> Integer {
    let %0: Integer
    let %1: Integer

    bb0(): {
        %0 = 99
        %1 = apply ({closure@0} as FnPtr) %0

        return %1
    }
}
```

After

```mir
fn {closure@1}() -> Integer {
    let %0: Integer
    let %1: Integer
    let %2: Integer

    bb0(): {
        %0 = 99
        %2 = %0
        %1 = %2

        return %1
    }
}
```

Listing 17: Forwarding closure reduction: the wrapper is eliminated and the inner call is exposed

Listing 17 illustrates a forwarding closure reduction: the wrapper body `{closure@0}`, which captures an argument and immediately delegates to another function, is eliminated by splicing its prelude and the inner call directly into the caller. The pass

processes bodies in call-graph postorder to ensure that callees are reduced before their callers. After transformation, each body is reclassified: a body that was not initially reducible may become so after its callees were inlined into it. The reducibility set grows monotonically, because the requirements of the two reducible forms guarantee that inlining a reducible callee cannot declassify the caller. Postorder traversal combined with monotonic reclassification ensures that a single pass achieves global maximality up to SCCs. This limitation is acceptable: the HIR/ANF does not generate recursive bindings, and a missed reduction within an SCC would delay simplification to the fixpoint loop rather than affect correctness. The pass is monotone: it replaces function calls with the callee's statements and never introduces new bodies or control flow.

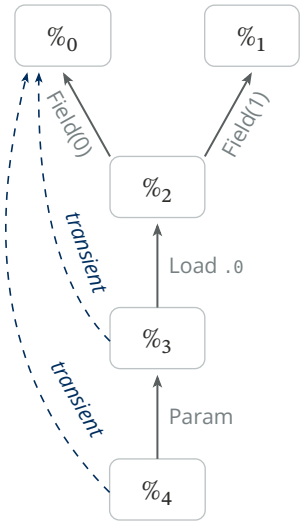
Forward Substitution (FS). Forward substitution (FS) replaces any place with the value that the place is known to hold. Where copy propagation, its counterpart, prioritizes speed, forward substitution prioritizes accuracy: it tracks values through projections, block parameters, and closure environments. To achieve this, the pass constructs a data dependency graph rather than using a single forward traversal or the dataflow framework. A graph with typed edges is better suited to representing nested access paths than a per-local lattice, and unlike a single pass, it can follow chains of projections across multiple definitions.

The data dependency graph records structural relationships between locals as directed edges, collected in a single pass through the body:

1. **Load.** A direct assignment from one local to another, equivalent to what copy propagation tracks, with additional projection tracking.
2. **Param.** A block parameter edge from the parameter local to the value each predecessor passes as an argument.
3. **Index.** A positional element within a tuple aggregate.
4. **Field.** A named field within a struct aggregate.
5. **ClosurePtr.** The function pointer operand during closure construction.
6. **ClosureEnv.** The captured environment operand during closure construction.

Non-structural relationships – binary operations, unary operations, and dynamic collection access – are excluded from the graph, because their results cannot be decomposed through projections. These six edge kinds cover all structural constructs in the MIR. Each edge carries a kind, which identifies the structural relationship, and an optional projection chain, which records the source-side access path on the target local. For example, a block parameter $\%_1$ that receives $\%_{2.0}.\text{foo}$ from a predecessor produces a Param edge from $\%_1$ to $\%_2$ with the projection chain $.0.\text{foo}$ attached to the edge.

```
1  fn {example@0}(%0: Integer, %1: Integer) -> Integer {     mir
2    let %2: (Integer, Integer)
3    let %3: Integer
4    let %4: Integer
5
6    bb0(): {
7      %2 = (%0, %1)
8      %3 = %2.0
9
10     goto -> bb1(%3)
11   }
12
13   bb1(%4): {
14     return %4
15   }
16 }
```



Listing 18: Data dependency graph for tuple projection and parameter propagation

Forward substitution constructs a transient graph by resolving the projection chain on every edge to its final destination before applying the results to the MIR body. Because the MIR is in SSA form and each local has exactly one definition, edge resolution is independent per edge and imposes no ordering constraints. Each edge target is followed through the graph: structural edges are matched against the projection, and load edges are followed transitively. If the resolved target is a constant, the edge is replaced with a constant binding; otherwise the edge is replaced with the resolved, potentially partial, target. The resulting graph therefore represents a transitive closure of the data dependency graph.

Algorithm 9: Dependency graph resolution

procedure Resolve(G, l, π)
  ▷ Follow Load edges transitively
  if G has a Load edge from l to l' with projections π' **then**
    $l \leftarrow$ Resolve(G, l', π').local
  end

  ▷ Check Param consensus across all predecessors
  if l has Param sources (graph edges or constant bindings) **then**
    $S \leftarrow \varnothing$
    for each Param source s of l **do**
      $r \leftarrow$ resolve s through full projection π
      if r is a cycle **then**
        skip this predecessor
      end
      else $S \leftarrow S \cup \{r\}$
    end
    if all $r \in S$ agree on value v **then**
      $l \leftarrow v$
    end
    else return Incomplete(l, π)
  end

  ▷ Base case: no projections remaining
  if $\pi = \varnothing$ **then**

```
25:     return Resolved(l) or Resolved(constant) if bound
26:   end
27:
28:   ▷ Match leading projection against graph edges
29:   (p, π′) ← split first element from π
30:   if G has edge from l matching p to target l′ then
31:     return Resolve(G, l′, π′)
32:   end
33:   return Incomplete(l, π)
34: end
```

The resolution algorithm (Algorithm 9) proceeds in three stages: transitive load following, predecessor consensus for block parameters, and projection matching against structural edges. Given a place consisting of a local l and a projection chain π, the algorithm follows structural edges in the data dependency graph to locate the value that the place holds. Load edges are followed transitively, as a load has exactly one source. If the local l is a block parameter, the algorithm checks whether all predecessors resolve to the same value after applying the full projection chain. Each projection in the chain is matched against its structural edge counterpart. A match continues resolution recursively until exhausted. When a resolution cannot follow an edge, it records the current local with the remaining projections as a partial result, rather than failing. This partial resolution is by design: removing chain indirection in the body allows other passes, such as dead store elimination (DSE), to eliminate intermediate aggregate constructions even when their components point to opaque structures.

The only source of complexity is block parameters. By definition of SSA, block parameters are the only locals that receive values from multiple predecessors, and predecessors connected by loop back-edges can introduce cycles. Resolution handles this by requiring consensus: only if all predecessors resolve to the same value is that value adopted, with recursive predecessors being excluded from the consensus check. If all predecessors are recursive, the resolution returns incomplete and propagates that result to the caller.

The pass is monotone: it replaces operands with values that the dependency graph proves equivalent and never introduces new instructions or locals.

Copy Propagation (CP). Copy propagation (CP) replaces uses of a local with the value that local is known to hold, where the value is either a constant or another local. The pass prioritizes speed over precision: rather than using the dataflow framework (Chapter 6.2), it performs a single forward traversal in reverse postorder. It records a mapping of local to known value, with load operations resolved transitively: if $\%_2 = \%_1$ and $\%_3 = \%_2$, both $\%_2$ and $\%_3$ are replaced with $\%_1$.

Algorithm 10: Copy Propagation

```
1: procedure Copy-Propagation(B)
2:   ▷ map from locals to known values
3:   V ← ∅
4:   for each block b in reverse postorder of B do
5:     for each parameter p of b do
6:       if all predecessors pass the same value v for p then
```

```
            V[p] ← v
          end
        end
        for each statement s in b do
          substitute uses in s via V
          if s is %_x = v where v is constant or V[v] is defined then
            V[%_x] ← resolved value of v
          end
        end
        substitute uses in terminator of b via V
      end
    end
```

At block boundaries, copy propagation propagates values through block parameters when all predecessors pass the same value for a given parameter. If any predecessor passes a different value, or if the predecessor's branch originates from an effectful terminator whose arguments are implicit, the parameter is left unresolved. Because the traversal is a single forward pass without fixpoint iteration, values carried along back-edges in loops are not discovered. This limitation is acceptable: forward substitution (FS) resolves recursive dependencies in alternation with copy propagation, and the outer fixpoint compensates for any remaining gap.

Copy propagation does not resolve through projections: an access such as $\%_{2.0}$ where $\%_2$ is a known tuple remains unchanged. This is the principal distinction from forward substitution, which builds a full data dependency graph and resolves through projections, block parameters, and closure environments. Because copy propagation is inexpensive and forward substitution handles the cases copy propagation cannot reach, the canonicalization loop alternates between them to balance speed against depth. Monotonicity follows directly: every substitution replaces a local with a value already present in the body, so the instruction count and local count are non-increasing.

Instruction Simplification (IS). Instruction simplification (IS) performs local algebraic simplification and constant folding on individual instructions. Like copy propagation (CP), it operates in a single forward traversal in reverse postorder, tracking a map from locals to known constants. The pass classifies each binary or unary instruction into one of three cases and applies the corresponding rewrite rule.

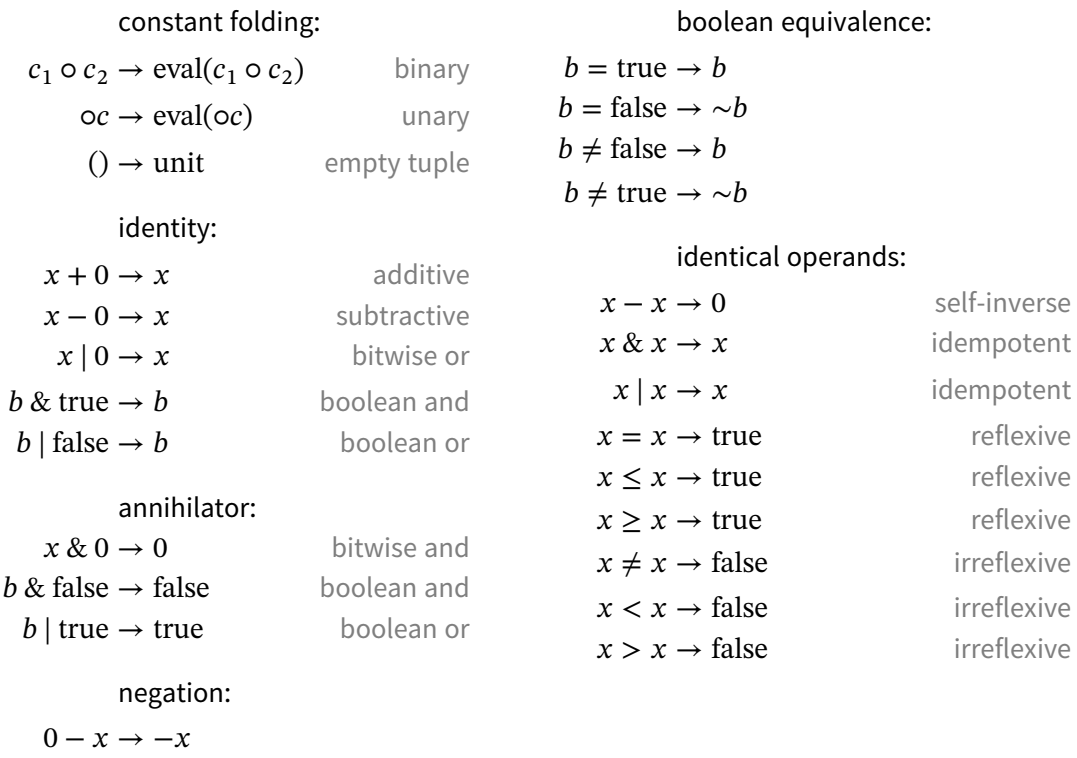


Figure 10: Instruction simplification rewrite rules

If both operands are constants, the operation is evaluated at compile time. A single constant operand triggers algebraic identities, annihilators, and boolean equivalences; identical operands trigger reflexive and idempotent rewrites. Constant results feed subsequent instructions through the same map. Figure 10 lists the complete set of implemented rewrite rules, where c denotes a constant, x an operand of any type, and b a boolean-typed operand. The pass also specializes empty tuple constructions to unit constants.

As with copy propagation, the single-pass traversal does not discover constants carried along loop back-edges. Monotonicity holds because every rewrite in Figure 10 reduces an instruction to a simpler or constant form; no rule introduces a new instruction or local. Chapter 10.1.4.3 discusses a potential unification of IS and CP through sparse conditional constant propagation (SCCP) [80].

CFG Simplification. CFG simplification reduces the number of basic blocks and simplifies control flow by applying local transformations. Unlike copy propagation (CP) and instruction simplification (IS), which traverse the body once, CFG simplification uses a worklist-based fixpoint: blocks are processed in roughly postorder, and predecessors of modified blocks are re-enqueued to propagate optimization opportunities backward through the graph.

The pass applies six transformations:

1. **Goto chaining.** Merge a block with its goto target when the target has a single predecessor, or when the target contains only no-op statements. The target's statements are appended to the current block and its terminator replaces the goto.
2. **Switch constant folding.** If the discriminant of a `SwitchInt` is a known constant, degenerate to a `Goto` targeting the matching arm.

3. **Switch degeneration.** If all arms of a `SwitchInt` point to the same target, with the same block arguments, degenerate to a `Goto` with said target.
4. **Redundant case removal.** Eliminate all switch cases whose target block and block arguments match the otherwise branch, when one is present.
5. **Target promotion.** If a switch arm targets an empty block – one containing only no-op statements and ending in a `Goto` – redirect the arm directly to that goto's destination, which bypasses the intermediate block.
6. **Otherwise-only degeneration.** Convert a `SwitchInt` with no explicit cases into a `Goto` to the otherwise target.

After each transformation, the pass recomputes reachability and marks newly unreachable blocks as dead immediately, which enables cascading. Any transformation may render a subgraph unreachable, reducing the predecessor count of downstream blocks and unlocking transformations that were previously inapplicable. For example, redundant case removal may leave a `SwitchInt` with only an otherwise branch, which degenerates to a `Goto` on the next iteration; the `Goto` reduces the target's predecessor count to one, which in turn enables goto chaining. This composability between the six transformations makes the worklist fixpoint effective. Once the worklist stabilizes, dead block elimination (DBE) compacts the block index space and SSA repair restores the single-definition property, as described in their respective sections. The composite pass is monotone: each transformation removes control flow or replaces a complex terminator with a simpler one, and no transformation introduces new blocks or edges.

SSA Repair. Because SSA repair is costly, all passes in the canonicalization loop are designed to preserve the single-definition property across their transformations. CFG simplification is the sole pass that cannot: merging two blocks that each define a local produces a block with two definitions. To ensure that no consumer observes a body with broken SSA, the repair pass runs inside CFG simplification itself, immediately after the core transformation completes. Consequently, CFG simplification preserves the SSA invariant as a composite pass.

This work implements the incremental reconstruction algorithm described by Fabrice Rastello and Florent Bouchez Tichadou (Eds.) [69:5.5.2], based on the dominance frontier. Rather than reconstructing SSA from scratch, the algorithm identifies only the locals with multiple definition sites and computes the iterated dominance frontier for those definitions. Block parameters are inserted at join points in the frontier, and definitions and uses are renamed to restore the single-definition invariant.

Unlike the other passes in the loop, SSA repair may introduce new locals and block parameters. Convergence is unaffected: the pass is idempotent on valid SSA, producing no changes when no violations exist. It activates only in response to violations that CFG simplification introduces and stabilizes in a single application.

Dead Store Elimination (DSE). Dead store elimination (DSE) removes assignments, block parameters, and storage statements whose values are never observed. The pass operates at statement granularity and does not address unreachable code, which is handled independently by CFG simplification.

A local is dead if no root use depends on it. A root use is a use consumed directly by a terminator or effect boundary, specifically the operand of a return, the discriminant

of a `SwitchInt`, or the resulting block parameter of an effectful terminator, rather than by another local's definition. Identifying dead locals is fundamentally a dataflow problem, but standard backward liveness (Chapter 6.2) propagates liveness for every use unconditionally. A dead cycle – a set of locals that depend on each other through block parameters but lack any root use – survives this analysis, because each member generates liveness for the others.

The correct formulation is strongly live variable analysis, a derived backward dataflow problem over a join powerset semilattice of locals. A local enters the live set in two cases: it is part of a root use, or it is used to compute a value already in the live set. Because liveness propagates only through data dependencies and the transfer function is monotone over the finite powerset lattice, per-block iteration over the CFG is not required. This work specializes the analysis further by operating over a dependency graph of $\text{def} \rightarrow \text{operand}$ edges rather than the CFG. The graph is traversed from root uses, following dependency edges to collect the full live set. Any local not reached is dead and subsequently removed [17:10.2.1].

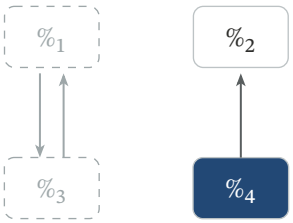
```mir
fn {example@0}(%0: Integer) -> Integer {
    let %1: Integer
    let %2: Integer
    let %3: Integer
    let %4: Integer

    bb0(): {
        goto -> bb1(0, %0)
    }

    bb1(%1, %2): {
        %3 = %1 + 1
        %4 = apply ({process@1} as FnPtr) %2

        switchInt(%4) -> [0: bb2(), 1: bb1(%3, %4)]
    }

    bb2(): {
        return 0
    }
}
```



Listing 19: Dead cycle: `%1` and `%3` depend on each other with no path to a root use

Listing 19 illustrates the difference. $\%_1$ and $\%_3$ form a cycle through the back-edge: $\%_1$ is used to compute $\%_3$, and $\%_3$ is passed back as $\%_1$ via the block parameter. Neither reaches a root use: the return uses a constant and the `SwitchInt` discriminant depends on $\%_4$, which traces to $\%_2$ independently. Standard backward liveness would classify both as alive, because each generates liveness for the other through use. Strongly live variable analysis correctly identifies that no root use is reachable from either and eliminates both.

Strongly live variable analysis is sound here because HashQL is referentially transparent: omitting an unobserved computation has no side effect. The MIR reinforces this by encapsulating all observable effects through effectful terminators, whose resulting

block parameters are always classified as root uses. Any local without a path to a root use can therefore be safely removed.

Algorithm 11: Dead store elimination

```
1:  procedure Dead-Store-Elimination(B)
2:      ▷ Phase 1: Build dependency graph and identify root uses
3:      G ← empty dependency graph
4:      L ← ∅          ▷ live set
5:      for each definition d with operands {o_1, ..., o_n} in B do
6:          for each o_i do
7:              add edge d → o_i to G
8:          end
9:      end
10:     for each root use r in B do
11:         L ← L ∪ {r}
12:     end
13:
14:     ▷ Phase 2: Propagate liveness from roots through G
15:     for each l ∈ L (expanding) do
16:         for each edge l → o in G do
17:             L ← L ∪ {o}
18:         end
19:     end
20:
21:     ▷ Phase 3: Eliminate dead definitions
22:     for each definition of local d in B do
23:         if d ∉ L then
24:             remove definition and corresponding block parameter arguments
25:         end
26:     end
27:     Dead-Local-Elimination(B)
28: end
```

Algorithm 11 outlines the procedure. The first phase constructs the dependency graph and identifies root uses. The second phase propagates liveness from roots through dependency edges until no new locals are reached. The third phase removes all definitions not in the live set, along with their corresponding block parameter arguments. Because removal turns the local index space sparse, dead load elimination (DLE) runs as a final step to compact the declarations, as described in its section.

The pass is monotone: it only removes statements and block parameters; the live set is a subset of the original, so no new definitions arise.

Dead Local Elimination (DLE). Each body in the MIR carries a typed declaration for every local, including provenance and type information. When dead store elimination (DSE) removes dead assignments, the local declarations themselves remain, turning a contiguous index space into a sparse one. Dead load elimination (DLE) complements DSE by removing those declarations: it scans the body for local references, removes any local that is never mentioned, and remaps the surviving locals to a contiguous sequence. This compaction is not necessary for correctness, but narrows the index space for downstream analyses, which allocate per-local data structures sized by the local count; density reduces both memory usage and time spent during analysis, as some

passes inspect each local. The number of locals removed is often significant, especially after inlining, which tends to inflate local counts. In the canonicalization pipeline, dead load elimination (DLE) runs as part of DSE rather than as a standalone pass. Because the pass only removes declarations, monotonicity is immediate.

Dead Block Elimination (DBE). Dead block elimination (DBE) follows the same pattern as dead load elimination (DLE), but targets basic blocks instead of locals. Any block not reachable from the entry block is pruned, and the surviving blocks are remapped to a dense index sequence. As with DLE, compaction is not required for correctness, but allows downstream analyses to skip unreachable blocks entirely; execution analysis in particular benefits, as it would otherwise allocate placement state for blocks that can never execute. In the canonicalization pipeline, DBE runs as part of CFG simplification rather than as a standalone pass. As with DLE, the pass only removes structure, so monotonicity follows directly.

7. Target Placement

Graph operations in HashQL follow a pipeline architecture in which closures such as filters and maps are executed once per vertex in a traversal. The underlying bi-temporal graph stores vertex data across heterogeneous backends that differ in both the data they source and the operations they can express: each backend may only support a subset of the MIR. This creates a tension between data locality, backend capability, and execution cost.

Target placement resolves this tension: given an optimized MIR body, it assigns each basic block to exactly one execution backend such that the total cost of computation and data transfer is minimized subject to capability constraints. Because the cost of a block depends on the targets assigned to its neighbors in the CFG, placement is forward-looking; a locally optimal assignment can be globally infeasible, as it may leave no feasible target for downstream blocks, which makes this an optimization problem rather than a per-statement dispatch rule.

This work defines a framework for execution targets: systems to which a subset of the MIR can be compiled for execution, and from which data may be sourced. The current implementation provides three targets:

1. **Interpreter.** The universal execution fallback. Due to its tree-walking nature, it is the slowest of the three, but can evaluate any MIR instruction.
2. **PostgreSQL.** The primary data source, lowering MIR fragments into SQL. Its declarative nature allows expressing most relational operations, but closures, function calls, and iteration have no SQL equivalent; additionally, some value shapes prevent unambiguous transfer, as elaborated in Chapter 7.2 and Chapter 8.
3. **Embedding.** Stores vector encodings of vertices and their individual properties separately from the graph. This target currently exists only in the analysis phase, as the underlying graph does not yet support a vector store (Chapter 10.1.2.5).

While a universal fallback is not strictly required, the interpreter's presence guarantees that every block has at least one feasible assignment, as described in Chapter 7.5. Each target defines a capability profile, which determines which statements it supports and their relative cost, as well as a transition profile, which specifies which backend-to-backend transitions are permitted. These transitions are asymmetric and directed, as further elaborated in Chapter 7.3. This asymmetry means a locally valid assignment may leave no feasible target for a downstream block.

The HASH graph distinguishes four vertex types: entities, entity-types, property-types, and data-types. Each graph operation pipeline is bound to exactly one vertex type, and its body closures operate on one vertex at a time. Data bound to that vertex, regardless of which backend stores it, is accessible through property paths: field projections in the MIR that the placement system can cost directly. Any operation that resolves to a different vertex, whether of the same type or another, constitutes a graph traversal and is modeled as an explicit graph effect.

Each graph effect carries its own pipeline, with its own vertex type binding and its own placement problem. Because graph calls cannot be recursive and their inputs and outputs are statically known, nested pipelines compose without interference: the current implementation dispatches each through the orchestrator, but the architecture permits fusing nested pipelines directly into the outer query when the inner pipeline is fully dispatchable to a single backend. This separation is what makes cost analysis tractable: the set of paths accessible within a single pipeline body is determined entirely by the bound vertex type, which is finite and statically known.

The vertex type also determines where each path originates. Relational properties, such as scalar fields and structural data, originate on PostgreSQL. Vector encodings originate on the embedding backend. Accessing a path on its origin backend requires no data transfer; accessing it on any other backend requires the orchestrator to fetch the data from the origin and deliver it to the consumer, at a cost proportional to the path's serialized size. This origin-based cost asymmetry is the data locality force that the cost model in Chapter 7.1 formalizes.

Placement takes as input an optimized MIR body produced by the transformation pipeline of Chapter 6 and produces per-block target assignments together with an island dependency graph. An island is a maximally connected region of basic blocks that share a target assignment. Compilation of assigned islands into backend-specific artifacts and their dispatch are the subject of Chapter 8.

7.1. Cost Model

Optimizing placement requires a metric. Two competing forces govern the cost of an assignment:

- **Computation Cost.** The relative expense of executing each MIR statement on a given backend, or unsupported if the backend cannot express the operation at all. Chapter 7.2 formalizes this component and attributes its weights.
- **Transfer Cost.** The expense of moving data between backends when a transition occurs along a CFG edge. Chapter 7.3 formalizes this component and attributes its weights.

Target transitions can only occur across CFG edges; the basic block is therefore the unit of assignment. Chapter 7.4 describes how blocks with mixed operation support are split to refine this granularity. The cost model minimizes the sum of these two forces across all blocks and edges. The model does not incorporate branch probability, but the size estimation used for transfer costs accounts for cardinality growth through loops. The result is a structural surrogate that ranks assignments by their estimated cost profile

rather than by measured runtime behavior. The solver therefore receives a single per-block metric that encodes both forces.

Transfer cost is a compound measure that decomposes into three sources, each with distinct target dependence:

- **Local transfer.** When a CFG edge crosses a backend boundary, all live-out locals must be serialized and transmitted to the successor. The cost is proportional to the number of live-out locals and their estimated sizes, and is zero when both sides of the edge share the same target.
- **Vertex path premium.** Vertex paths load data from a backend determined by the path's origin, not by the successor's assignment. If the block is assigned to a target that is not the origin for an accessed path, a transfer premium proportional to the path's estimated size is incurred. This component depends on the block's own assignment, not on its neighbors.
- **Static transition overhead.** Each target-to-target pair incurs a near-constant overhead for serialization and deserialization, independent of data volume. The overhead may vary by terminator kind.

To participate in the optimization problem, each cost component must be attributed to either nodes or edges in the CFG. For most components, this follows naturally: computation cost is a cumulative measure of the statements and terminators in a block and therefore bound to the node. Local transfer and static transition overhead depend on the source and destination targets and therefore bound to the edge. Vertex path premiums, however, do not fit either attribution cleanly. They depend not on which targets a transition connects, but on which target the block itself is assigned to and which paths it accesses. Three attribution strategies expose distinct trade-offs between granularity and tractability.

7.1.1. Edge-Based Attribution

The "vertex-as-local" strategy treats the vertex as a regular local. Because the vertex local is live throughout the body, the transfer cost computation charges the full estimated vertex size at every backend transition, negating the usefulness of a heterogeneous approach and compiler altogether. An interpreter block performing pure arithmetic, with no vertex access, still pays the transfer cost for vertex data it never reads. The failure is one of granularity: the model does not distinguish which paths a block actually accesses, so every transition pays for the entire vertex.

The "vertex-dead" strategy suppresses the vertex local from liveness entirely, leaving edge transfer costs at zero for vertex data. This severely under-approximates: it assumes that data transfers between backends at no cost. Because most of the substantial data in a pipeline body originates from the graph, not from the program, the resulting cost model cannot account for the dominant transfer component.

The "path-at-edge" strategy refines the first by tracking individual vertex paths through a dataflow lattice. A statement or terminator that accesses a vertex path marks it as required; because liveness propagates backward, the path is then live from the body's entry up to the accessing block. The cost is charged at edges that cross a backend boundary, and only for paths that are actually live at that edge, with same-backend

edges paying nothing. This resolves the granularity problem of "vertex-as-local", but exposes a deeper issue: once a path is marked as required, there is no mechanism to discharge it. Theoretically, the path could be discharged when execution reaches the origin backend, since the origin fetches from its own store and requires no transfer. But determining whether a block is on the origin backend requires knowing its target assignment, which depends on the cost model itself, creating a circular dependency. Without discharging, the model faces two choices: assume every path is immediately satisfied (equivalent to "vertex-dead") or assume no path is ever satisfied. The latter is the only viable option, but it over-charges in both directions: edges leading into the origin backend are charged as if they must ship data to it, and edges leading away from the origin carry the cost onward to intermediary blocks that may not need it.

Consider a block on the embedding backend that loads vector encodings. Because the path is never discharged, all edges leading to the embedding block are charged as if they must ship vector data to it. In reality, the embedding backend fetches the data from its own store; no transfer occurs. The over-approximation biases the solver against placing intermediate blocks on backends that would otherwise be beneficial.

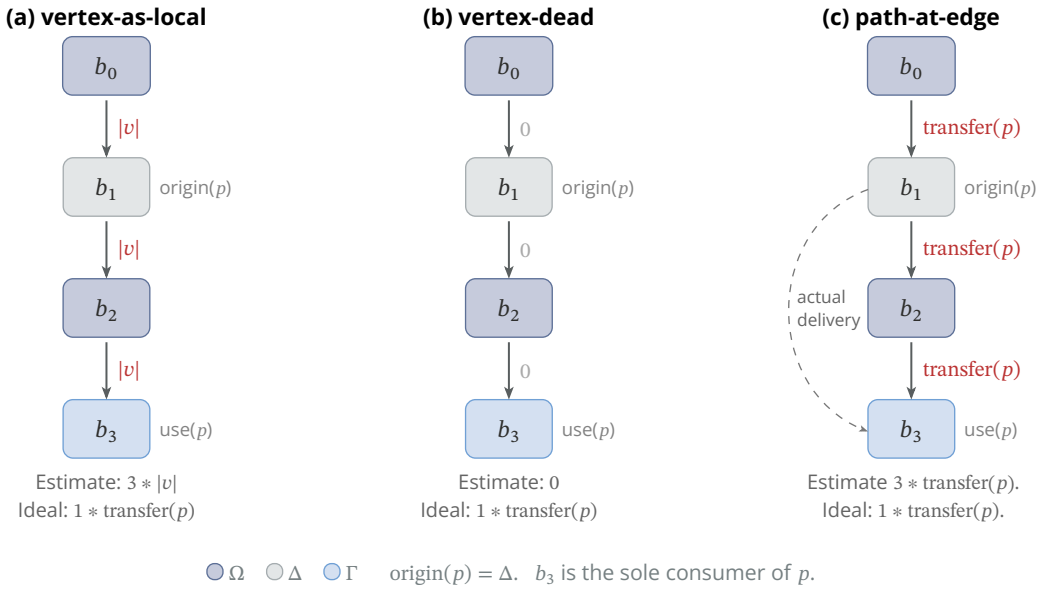


Figure 11: Edge-based vertex path attribution strategies.

7.1.2. Block-Based Attribution

All three edge-based attribution approaches fail for the same reason: vertex path data does not flow along the CFG. Locals are scoped, small, and directly tied to control flow; they have well-defined points where they are created and consumed, making edge-based costing natural. Vertex data has none of these properties. If three islands, each on a distinct target Ω, Δ, and Γ, are connected as $\Omega_1 \to \Delta_1 \to \Gamma_1$, and Γ_1 requires data originating on Ω, routing the data through Δ_1 would require serialization and deserialization at each hop, which is both prohibitively expensive and potentially infeasible if Δ restricts the types of data it can carry. The delivery path of graph data therefore does not follow CFG edges: the runtime fetches directly from origin to consumer, bypassing intermediate backends.

In the base case, data is fetched from the origin and delivered to the consuming block; if a preceding island on the same target has already fetched the path, the data may be reused. The cost model conservatively assumes the base case, charging the full premium regardless of potential reuse. The premium reflects data volume only. Each path has exactly one origin and the data must always be retrieved, regardless of assignment. Any fixed overhead of the origin-local lookup is constant across all candidate assignments and does not influence relative ranking; it is omitted from the model. Any fixed overhead of remote delivery applies precisely when $\text{origin}(p) \neq \Omega$, the same condition the premium charges. Because this overhead adds equally to every non-origin candidate for a given path, it does not alter their relative ranking and is therefore omitted.

We encode this insight by charging the vertex path premium at the block, not at the edge. If $\texttt{bb}_n$ accesses a path p whose origin is Δ, and $\texttt{bb}_n$ is assigned to a different target Ω, the data must originate from Δ regardless of the CFG path leading to $\texttt{bb}_n$. Sharing data from a predecessor that has already fetched it is an optimization that reduces the true cost, but the base case is the full fetch; any sharing is a discount from that base. The per-block premium is therefore a structural upper bound on the per-block component of the true transfer cost: it may overcount when multiple same-target blocks share a fetch, but it never charges less than a single block's actual need.

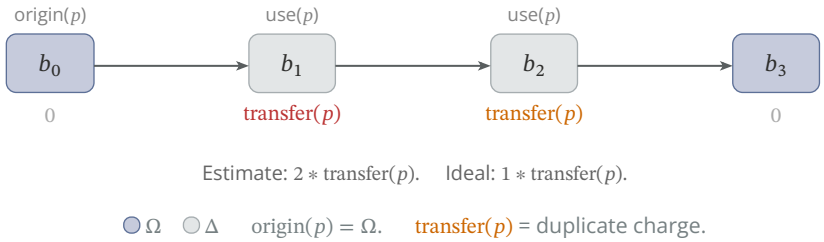


Figure 12: Block-based attribution with separable over-approximation.

7.1.3. Objective Formulation

Block-side charging over-approximates in two ways. The first is within a single block: if multiple statements in $\texttt{bb}_n$ access the same path p, the premium is charged once per statement. The resolution is straightforward: charge once per path per block, recording path presence rather than access count. Let $\text{paths}(\texttt{bb}_n)$ denote the set of distinct vertex paths accessed by any statement or terminator in $\texttt{bb}_n$, and $\text{origin}(p)$ the backend that stores path p. The intra-block premium for assigning $\texttt{bb}_n$ to target Ω is:

$$\text{premium}(\texttt{bb}_n, \Omega) = \sum_{p \in \text{paths}(\texttt{bb}_n)} \mathbf{1}[\text{origin}(p) \neq \Omega] * \text{transfer}(p) \tag{7}$$

where $\text{transfer}(p)$ is the estimated serialized size of p. This deduplication is exact and introduces no approximation.

The second over-approximation is across blocks. Consider two consecutive blocks $\texttt{bb}_1$ and $\texttt{bb}_2$, both assigned to target Ω, each accessing the same path p with $\text{origin}(p) \neq \Omega$. Under the ideal cost model, the data is fetched from $\text{origin}(p) = \Delta$ once and $\texttt{bb}_2$ reuses it at no additional cost. The ideal charging granularity must therefore be coarser than per-block. The naive candidate is per-island, but this over-approximates as well. Consider a

CFG subgraph $\texttt{bb}_0 \to \texttt{bb}_1 \to \texttt{bb}_2$ with an additional entry $\texttt{bb}_3 \to \texttt{bb}_2$, all blocks on target Ω, where $\texttt{bb}_1$ accesses a path p_1 with origin Δ. If the island is only entered through $\texttt{bb}_3$, the island-level charge records p_1 as required, even though $\texttt{bb}_1$ is unreachable from $\texttt{bb}_3$ and p_1 is never accessed, a concrete over-approximation. The required paths are therefore not island-specific but entry-point-specific: what matters is the set of blocks reachable within Ω_1 from each entry point, $R(\texttt{bb}_n, \Omega_1)$. Formally, let P denote the set of all accessed paths, T the set of available targets, and $x_{\texttt{bb}_n}$ the assignment of block $\texttt{bb}_n$. The ideal cost is:

$$C_{\text{ideal}} = \sum_{p \in P} \sum_{\Omega \in T \setminus \text{origin}(p)} \sum_{e \in \text{entries}(\Omega)} \text{transfer}(p) * \mathbf{1}[\exists \texttt{bb}_n \in R(e, \Omega) : p \in \text{paths}(\texttt{bb}_n)] \quad (8)$$

The indicator $\mathbf{1}[\exists \texttt{bb}_n \in R(e, \Omega) : p \in \text{paths}(\texttt{bb}_n)]$ makes C_{ideal} naturally submodular: the marginal cost of adding a block to a reachable set is zero if any block already in the set accesses p, and $\text{transfer}(p)$ otherwise. Unlike standard pairwise submodular functions, the reachability requirement within islands means the submodular coupling operates over higher-order cliques rather than pairwise terms.

C_{ideal} captures only the shared-fetch premium. The full ideal placement objective includes per-block computation costs (statement and terminator) and pairwise transition costs along CFG edges. Because branching terminators may target the same successor block from multiple arms, each carrying values of different narrowed types and therefore different transfer costs, the CFG is a directed multigraph $G = (B, E, \text{src}, \text{dst})$, where B is the set of basic blocks, E the set of concrete control-flow edges, and src, dst the operations that yield the source and destination of an edge. The full ideal placement objective is:

$$E_{\text{ideal}}(x) = \sum_{\texttt{bb}_n \in B} u\big(\texttt{bb}_n, x_{\texttt{bb}_n}\big) + \sum_{e \in E} t_e\big(x_{\text{src}(e)}, x_{\text{dst}(e)}\big) + C_{\text{ideal}}(x) \quad (9)$$

where u is the per-block computation cost (statement and terminator combined) and t_e is the pairwise transition cost for edge e. Disregarding C_{ideal}, the remaining terms form a classical pairwise energy minimization problem[a], albeit with a non-metric cost. The transition term t_e introduces asymmetry: a transition from one execution target to another does not guarantee the inverse, as shown in Chapter 7.3, where PostgreSQL permits transition out but not in. A metric cost function requires symmetry, which the transition matrix violates. C_{ideal} adds the higher-order submodular structure from shared fetch on top of this pairwise base. The problem is also multi-label: this work uses $k = 3$ labels, with the architecture permitting expansion to arbitrary k without exponential growth in computation time. The conjunction of these properties, non-metric asymmetric pairwise terms, higher-order submodular cliques, multiple labels, and cycles in the CFG, is what makes E_{ideal} intractable.

Computing C_{ideal} requires knowing the final set of $R(e, \Omega)$ for each entry point, but these reachable sets depend on the island structure, which depends on the assignment, which is what the cost model is supposed to determine. An alternative approach inverts the perspective: instead of computing reachability from entries, each node inspects its

[a] Given a graph with discrete labels per node, energy minimization finds the labeling that minimizes the sum of per-node and pairwise edge costs.

predecessors. In a DAG, blocks can be processed in topological order, guaranteeing that each block is visited only after its predecessors. If all predecessors are on the same target and already cover a given path, the block can omit its own charge; otherwise it charges the path and records it for its successors. This fails in two distinct ways.

First, it distorts the cost model at non-homogeneous join points. If a join point does not have all predecessors agreeing on which paths have been charged, the cost must be issued regardless, which leads to an over-approximation. While this over-approximation charges less than a naive model would, it is by definition non-uniform and disadvantages join points specifically, which makes it more harmful than any uniform scaling solution. The solver is biased toward splitting at joins even when co-location would be cheaper.

Second, the approach fails in the presence of loops. No topological order exists for a strongly connected component, and unlike in the case of Chapter 6.3.1, loop breakers are unsound here, as they would violate valid transitions between edges. No canonical first block can therefore be determined, and any solution must revert to per-block charging for the entire SCC. The result is skewed in the same way as join points, but more severely: loops operate at a larger scale, and the fallback affects every block in the component. This is particularly disadvantageous because loop regions benefit from cost sharing the most. Transfer costs accumulate with each iteration, so any co-location decision that avoids a backend transition pays off multiplicatively.

Since neither the ideal cost nor the predecessor-based alternative is computable before solving, the question is whether any polynomial-time method can optimize E_{ideal} with global guarantees. The answer rests on established hardness results in two adjacent fields: submodular structure over higher-order cliques, and multi-label energy minimization with arbitrary asymmetric, non-metric pairwise costs on cyclic graphs.

The submodular structure over higher-order cliques of C_{ideal} has direct analogues in energy minimization on markov random fields (MRFs). The shared-fetch indicator $\mathbf{1}[\exists \texttt{bb}_n \in R(e, \Omega) : p \in \text{paths}(\texttt{bb}_n)]$ can be recast as a set function over $(\text{path}, \text{entry point})$ pairs, which corresponds to what Stefanie Jegelka and Jeff A. Bilmes [35] define as a cooperative cut with submodular edge weights: the cost of adding an element to a set depends on which elements are already present. Cooperative cuts are NP-hard even for nonnegative submodular costs, but the submodular structure admits approximation algorithms for the binary case. Separately, Vladimir Kolmogorov [45] provides an exact framework for minimizing sums of submodular functions over binary variables via submodular flow [35].

To extend the submodular problem to $k > 2$, two approaches exist. Stefanie Jegelka and Jeff Bilmes [34] extend cooperative cuts to the multi-label case via move-making: at each step, one label is selected and every node chooses between its current label and the selected one, reducing the multi-label problem to a sequence of binary subproblems. The result is a local minimum without global guarantees. Alpha-expansion follows the same move-making pattern but applies more broadly: it preserves the submodular structure in each binary subproblem, and for metric pairwise costs, Y. Boykov, et al. [13] show that it provides a bounded approximation guarantee for the pairwise component, with a factor of 2 in the uniform-metric case. This guarantee requires the metric condi-

tion, which the placement problem does not satisfy: the transition matrix is asymmetric and some transitions are disallowed entirely [13, 34, 35, 45].

The other relevant field is multi-label energy minimization with arbitrary asymmetric pairwise costs on cyclic graphs. Mature pairwise solvers exist: linear programming (LP) relaxations solved via tree-reweighted sequential message passing (TRW-S), tightened with cycle consistency, and augmented with quadratic pseudo-Boolean optimization (QPBO)-based fusion moves handle the pairwise case without metric assumptions.

These two fields have developed largely in isolation. Several methods attempt to combine higher-order clique costs with multi-label pairwise optimization, but each fails on at least one property that C_{ideal} requires:

- **Order reduction** can transform higher-order binary MRFs into equivalent pairwise ones with preserved minima, but is limited to $k = 2$. For $k > 2$, move-making algorithms such as alpha-expansion reduce each step to a binary subproblem, which can then be order-reduced, but the outer loop yields only local minima, and without the metric condition the approximation guarantees that bound their quality do not apply [33].
- P^n **Potts** admits efficient expansion and swap moves via s-t min-cut, but requires a uniform penalty whenever any nodes in a clique disagree on their label, regardless of which labels are involved. The shared-fetch cost violates this: its cost depends on which target a block is assigned to relative to the path's origin, and diminishes as the reachable set on the same target grows [43].
- **Generic Cuts / Multi-label Generic Cuts** computes optimal solutions for 2-label submodular clique potentials, with Multi-label Generic Cuts extending this to $k > 2$ by encoding multi-label potentials as 2-label higher-order problems. Both require the clique potentials to satisfy specific submodularity conditions. No existing work establishes whether asymmetric, non-metric transition costs admit such an encoding, and proving this is outside the scope of this thesis [6, 7].
- **Fusion moves with QPBO** reduce each step to a binary subproblem between the current and a proposed labeling. QPBO can solve non-submodular pairwise binary problems with partial optimality, but the outer move-making loop yields only local minima, and the higher-order clique structure of shared-fetch costs has no direct pairwise representation [46, 51].
- **sequential reweighted message passing (SRMP)** extends the TRW-S message-passing framework from pairwise to higher-order graphical models and provides convergent lower bounds on cyclic graphs, but the primal assignments extracted from these bounds are not guaranteed optimal [44, 45].

The difficulty is the conjunction of four properties: higher-order submodular clique costs from shared fetch, an asymmetric non-metric cost function, multiple labels, and cycles in the CFG. On the pairwise level alone, the hardness is already established: Julia Chuzhoy and Joseph (Seffi) Naor [16] prove that no constant-factor approximation exists for metric labeling unless P=NP. The metric labeling problem is a special case of the general non-metric problem, so this lower bound carries over. The use of higher-order submodular cliques through islands compounds this: it introduces a non-separable group cost that depends on the full assignment, outside the pairwise domain. The current implementation uses $k = 3$ targets, but the architecture is designed for

arbitrary k; the cost model must therefore be validated against the general case, not constrained by the current label count [16].

Generic exact approaches such as integer linear programming (ILP) and branch-and-bound (BnB)[b] can in principle enumerate all assignments, and BnB is used inside the solver for looping regions (Chapter 7.5), but full enumeration is impractical for the instance sizes and compilation-time budgets this system targets.

7.1.4. Separable Approximation

We therefore adopt a separable approximation that charges per block rather than per island. This eliminates the higher-order term and the submodular structure entirely; what remains is standard pairwise energy minimization. Worst-case inapproximability persists due to looping regions, but the instances produced by real queries are small enough for the solver presented in Chapter 7.5 to find exact or near-exact assignments using branch-and-bound. Under this model, every block pays the full transfer premium for each accessed path whose origin differs from the block's assigned target, with no cross-block deduplication. The resulting error is relative to C_{ideal}: it overcounts along same-target chains where blocks share a fetch, but may undercount when a single block is reachable from multiple island entries, since C_{ideal} charges once per entry while the per-block model charges once per block. The overcount dominates in practice: same-target chains with shared paths are common, while multi-entry convergence on a single accessed block is rare. The per-block cost on target Ω is:

$$\begin{aligned} \text{stmt}(s, \Omega) &:= \text{cost of statement } s \text{ on target } \Omega \\ \text{term}(\texttt{bb}_n, \Omega) &:= \text{cost of terminator in } \texttt{bb}_n \text{ on target } \Omega \\ \text{premium}(\texttt{bb}_n, \Omega) &:= \text{path transfer premium for } \texttt{bb}_n \text{ on target } \Omega \\ c(\texttt{bb}_n, \Omega) &= \sum_{s \in \texttt{bb}_n} \text{stmt}(s, \Omega) + \text{term}(\texttt{bb}_n, \Omega) + \text{premium}(\texttt{bb}_n, \Omega) \end{aligned} \tag{10}$$

The per-block cost $c(\texttt{bb}_n, \Omega)$ over-approximates non-origin assignments. If five blocks on Ω each access a path p with $\text{origin}(p) \neq \Omega$, the model charges $5 * \text{transfer}(p)$, but the runtime fetches the data once; the true cost is $1 * \text{transfer}(p)$. Despite this inflation, the premium preserves the structural properties required for solver correctness. For a given path p, the premium $\text{transfer}(p)$ is origin-independent: it is identical across all targets that are not p's origin. Pairwise comparisons among non-origin candidates are therefore undistorted by any single path's premium, and their relative ranking is preserved. Between origin and non-origin candidates, however, the premium is not neutral: it is zero for the origin and positive for every alternative. This asymmetry is deliberate. It reflects the data-locality advantage that a backend holding the path already provides, and biases the solver toward the origin as the default. The bias is conservative, not absolute: statement base costs and eligibility constraints can still outweigh the premium, so the data-locality signal does not drown out other factors in the objective.

[b] Branch-and-bound searches the assignment space exhaustively. Any subtree whose lower-bound cost exceeds the best complete solution found so far is pruned.

A natural attempt to move closer to C_{ideal} is partial deduplication: if a block has a single same-target predecessor outside an SCC that has already been charged for a path, deduct the premium. This is exact for such chains, since the non-SCC requirement guarantees topological processing order. The deduction fails everywhere else: join points have multiple predecessors with no single source of truth, and blocks inside strongly connected components have no topological predecessor at all. The approach suffers from the same skew as the inverse perspective discussed above, producing an inconsistent approximation where DAG chains are costed precisely while join points and loops are over-charged by the full premium. The solver may prefer a different target for a loop entry block solely because the loop-interior premium is higher, not because the target is cheaper. Uniform over-approximation avoids this: the same rule applies to every block in every region of the CFG, without switching between exact and over-approximate regimes. The error magnitude still varies with reuse structure, but no structural artifact is introduced at the boundary between DAG chains and the rest of the program.

Partial deduplication has a second cost: it breaks separability. If a block's premium depends on whether its predecessor has already been charged for the same path, the unary term $c(\mathtt{bb}_n, \Omega)$ is no longer a function of $\mathtt{bb}_n$'s target assignment alone; it also depends on the predecessor's assignment and charge state, which destroys the clean decomposition into per-node and per-edge terms over target labels. The uniform over-approximation avoids this. Because every block's premium depends only on its own assignment and the paths it accesses, the placement problem decomposes cleanly into unary and pairwise terms:

$$E(x) = \sum_{\mathtt{bb}_n \in B} c\left(\mathtt{bb}_n, x_{\mathtt{bb}_n}\right) + \sum_{e \in E} t_e\left(x_{\text{src}(e)}, x_{\text{dst}(e)}\right) \tag{11}$$

where c is the per-block cost (the unary term) and t_e is the transition cost for edge e (the pairwise term), with no third-node dependency. This formulation maps directly to Partitioned Boolean Quadratic Programming (PBQP), developed for register allocation in compilers. Both problems share the same abstract shape: a graph with a small label set per node and pairwise costs on edges, with the objective of minimizing total cost. PBQP admits R1 and R2 reductions that collapse degree-1 and degree-2 nodes by folding their costs into the cost vectors of neighboring nodes, often solving the problem exactly without search. Whether reductions suffice depends on the graph structure; any irreducible remainder requires heuristic selection. Because PBQP is an additional solving step rather than a replacement for the core solver, this work builds the solver independently first and defers PBQP integration to future work (Chapter 10.1.1.3) [28, 42, 74].

7.2. Statement Placement

Given the cost model presented in Equation 11, statement placement determines the non-premium components of Equation 10: the per-statement cost $\text{stmt}(s, \Omega)$ and the per-terminator cost $\text{term}(\mathtt{bb}_n, \Omega)$. Because terminators are subject to the same eligibility and costing rules as statements, the term "statement" covers both throughout this section.

The cost of a statement depends on the target. Each execution target implements a different computational model with its own type system, operational semantics, and serialization format, supporting only a subset of the instructions expressible in the MIR. Statement placement must account for these constraints: for each target, a statement is classified as either supported, receiving a finite cost that reflects the target's execution overhead relative to other backends, or unsupported, receiving a cost of $+\infty$.

Eligibility is a whole-program property, as each instruction depends on prior computations through its operands. A statement is supported only if the operation itself belongs to the target's expressible subset and all of its operands are computable on that target. This property is contagious: an unsupported operation renders every downstream consumer of its result ineligible, as the target cannot produce the value. The analysis is conservative. Operands that may be transferable to a target are rejected if they cannot be computed on the target. While this sacrifices some optimization opportunity, it guarantees correctness, and allows the producer-consumer propagation to be modeled as a dataflow problem. The analysis uses the dataflow framework described in Chapter 6.2 as a forward must-analysis over a meet powerset lattice. Forward, because operands of an instruction influence the computability of the resulting local, so information flows from definitions to uses. Must, because a local is eligible only if it is computable on every path that reaches it; at join points, the analysis takes the intersection (meet) of incoming states rather than the union. The lattice is the powerset of locals ordered by inclusion, with top representing all locals supported. Each unsupported operation narrows the set, either through the contagion property or through disagreement at a join point, until a fixpoint is reached, yielding the eligibility domain for each target.

The computability check operates at different granularities depending on the kind of operand, with function arguments requiring special treatment. Ordinary locals fully participate in the dataflow computation and are either computable on a target or not. The captured environment is checked per-field: computability is determined for each element of the environment tuple independently. Marking the entire environment local in the dataflow would be correct but unnecessarily conservative, because the environment is by definition never used as a value, only projected into, as each projection maps to a captured local. Per-field granularity allows individually representable captures to remain eligible even when other fields in the same environment are not. The vertex argument is handled by backend-specific predicates rather than the dataflow lattice. Each target admits only projections into paths stored on its native backend. Admitting all projections regardless of origin would be correct, as the path transfer premium already accounts for non-origin access cost, but restricting to native paths reduces the solver's search space without discarding placements that would yield a benefit. Chapter 7.4 details how the premium attaches to individual blocks.

For derived locals inside the body, a less conservative analysis could extend eligibility to include operands whose types are transferable across a backend boundary, regardless of whether they were computed on the target. While correct in principle, this creates a perverse incentive: it shifts the criterion from "the target must support the computation" to "the target can receive the value." The solver is then incentivized to take the transfer penalty for small operations, offloading cheap arithmetic to another backend instead of computing it where the data originated. The cost asymmetry between

targets can outweigh small transfer costs; the search space grows for cases that are unlikely to yield better placements. A refinement requires that at least one operand be computable on the target while the rest may be transferable. While this appears sound, the property cannot be expressed in the current lattice formulation: the meet-semilattice tracks per-local membership, not per-operand roles within a statement, so at a join point the distinction between computable and transferable collapses to the conservative check without enriching the abstract domain. Computation should be co-located with the data it operates on, the same principle that motivates restricting vertex projections to native paths; the conservative analysis enforces this directly.

The concrete eligibility rules and cost values differ across the three execution targets. Table 3 summarizes the per-operation costs and restrictions. PostgreSQL supports most relational operations but is constrained by its type system and declarative execution model. The embedding backend admits only vector encoding projections, as it serves as a validation target without an execution backend. The interpreter is the universal fallback and supports the full instruction set.

Operation	**Interpreter**	**PostgreSQL**[c]	**Embedding**
Load	8	4[d]	4[d]
Binary operation	8	4[e]	×
Unary operation	8	4[d]	×
Aggregate (non-closure)	8	4[d]	×
Closure construction	8	×	×
Function application	8	×	×
Input	8	4	×
Nop	0	0	0
`Goto`	8	4[d]	4[d]
`SwitchInt`	8	4[d]	4[d]
`Return`	8	4[d]	4[d]
`GraphRead`	8	×	×
`Unreachable`	0	0	0
Numbers denote cost (lower is better). ×: ineligible.			

Table 3: Statement and terminator costs per execution target.

7.2.1. PostgreSQL

PostgreSQL is the most capable of the specialized backends and the most constrained. Its computational model is declarative SQL: every operation must be expressible as a SQL expression or clause. This restricts both the types of values the backend can represent and the operations it can perform.

[c] Additionally requires serialization-safe result types. Function pointer constants are rejected.
[d] If all operands are computable.
[e] If all operands are computable, with an additional equality safety check for $=/\neq$.

Value representation is the first source of ineligibility. SQL's native type system does not accommodate the full range of HashQL types, and most vertex data is stored in JSONB documents, which lack native representations for several of those types. The compiler must therefore collapse the value representation into a JSON-compatible form.

The most direct alternative is a tagged encoding, where each value carries a disambiguating type tag alongside the payload, similar to the syntax developed for J-Expr as part of the Großer Beleg. This would allow lossless round-tripping at the cost of increased transfer size. The format is infeasible, however, because vertex data in the graph store is already stored as untagged JSONB. Adopting a tagged representation would require converting the existing data at the boundary, which in turn requires fetching type information to guide reconstruction. Any such conversion would need to happen both inside the runtime and inside PostgreSQL at query time, because values produced by either side must use the same representation to be comparable. A custom extension could handle this efficiently but is incompatible with the deployment requirements of the database. Stored procedures are available but amount to tree-walking the JSON structure in SQL, which is infeasible at scale. Migrating the data layer to a tagged format is outside the scope of this work [11].

Without a tagged format, the values of structurally distinct HashQL types collapse into overlapping JSON forms (Chapter 8.2). For concrete types the overlap is benign, as the MIR retains full type information and can reconstruct the original type after deserialization. Union types break this assumption. Because the semantic layer is collapsed, a union whose members map to the same JSON form creates an ambiguity that cannot be resolved from the value alone. The following patterns disqualify a value from serialization:

1. A tuple and a list in the same union. Tuples are width-invariant with their fields joined pointwise during type simplification, so type-checked unions guarantee that distinct tuples always differ in width and remain distinguishable. A list type in the same union with a tuple has no such guarantee and leads to a rejection [11].
2. A struct and a dictionary in the same union. A struct and a dictionary are always disqualifying, as both collapse to the same JSON form. The same applies to an open struct in the same union as a closed struct. Multiple closed structs do not lead to a rejection, as they are width-invariant and joined pointwise during type simplification; the key set therefore allows for disambiguation [11].
3. An opaque type in any union. The semantic collapse erases nominal identity: the opaque wrapper is lost in JSON and replaced by the inner value. We cannot guarantee that the underlying type is not itself a union member; the value is therefore rejected. This is conservative, as the inner type is not necessarily a member of the union, but the rejection is sound in all cases.

Separate from union ambiguity, some HashQL types cannot receive a JSON representation at all. Dictionaries with non-string keys have no encoding, as JSON object keys are strings by definition. Closures present a less obvious constraint: while the function pointer itself is an integer that could be serialized, the fat pointer design (Chapter 6.1) means closures also carry an environment tuple of captured values. These values may themselves be serialization-unsafe, so closures are rejected as a whole, as the type

system alone does not determine which variables have been captured. The full set of representation rules is given in Chapter 8.2. Because PostgreSQL has no incoming transitions from other backends (Chapter 7.3), these constraints apply only to values produced within the body, not to values that could hypothetically be transferred in.

Native SQL types do not resolve these limitations. SQL row types require explicit, static declarations: every type used in a query must be created before execution and destroyed afterward. HashQL's type system generates types dynamically as part of inference and simplification, so the set of types a query operates on is not known until compilation. Each query would therefore turn from a pure read query into a sequence of type creation, computation, and cleanup, which transforms the execution model from declarative to procedural. Even if this overhead were acceptable, the mapping is incomplete: union types have no SQL equivalent, dictionaries with arbitrary key types have no representation, tuples have no native encoding, and structs would require dynamically creating a row type for each distinct struct shape encountered during execution, because open structs make it impossible to enumerate the full set of struct types ahead of time.

Representational collapse affects comparison semantics as well as serialization. An equality check between two values that collapse to the same JSON form but originate from distinct HashQL types produces incorrect results: the comparison evaluates to true where the MIR semantics require false. Consequently, the analysis must verify that both operands of an equality or inequality comparison are free of representational collisions before admitting the statement to PostgreSQL.

Beyond representation, PostgreSQL is restricted to operations expressible in declarative SQL. Arithmetic, logic, comparisons, and aggregate construction all have direct SQL equivalents. Function calls do not: SQL has no mechanism for invoking a function pointer whose target is determined at runtime. Because aggressive inlining (Chapter 6.3.1) resolves most call sites before placement, the remaining call sites are those the inliner could not resolve: recursive functions, call chains that exceed the inlining depth bound, or indirect function pointers that could not be resolved during canonicalization (Chapter 6.3.2). These are unconditionally rejected.

PostgreSQL admits every terminator whose operands are themselves admitted. Effectful terminators are excluded, as they require runtime coordination beyond what a single backend can provide; their results are therefore also ineligible on PostgreSQL. Chapter 10.1.2.2 discusses how a future iteration could fuse effectful terminators that operate on the graph into PostgreSQL joins under the correct circumstances.

7.2.2. Embedding

The embedding backend exists only in the analysis pipeline. No execution backend is implemented, as the underlying graph store does not yet support a dedicated vector store (Chapter 10.1.2.5). The placement analysis models it as a minimal target, sufficient to exercise the solver against a heterogeneous three-target configuration. To avoid encoding premature assumptions into a placement strategy for a store that does not yet exist, the supported set of statements is kept deliberately small. Only loads from entity projections that resolve to the vector encoding path are admitted. The environ-

ment is non-transferable, and no computation beyond loading a value is supported. Control flow terminators are admitted if their operands are computable, but effectful terminators are not.

7.2.3. Interpreter

The interpreter is the universal fallback: it does not support any data source but evaluates the full instruction set, which guarantees that the solver can find a valid assignment for every block. Due to the tree-walking design, it is slower than any specialized backend and is used whenever no specialized backend can express a computation or where deferral to one would yield prohibitively large transfer costs. Because the interpreter executes within the runtime directly, it is the only backend currently able to execute effectful terminators.

7.3. Transfer Cost

Transfer cost materializes the pairwise term t_e from Equation 11. As with statement placement, the cost is determined upfront, before the solver runs, and stored in a transition matrix. For each edge $e \in E$, the matrix enumerates every target pair (Ω_i, Ω_j): k^2 entries for k available targets. Each block carries a uniform set of candidate targets, determined after the block splitting pass in Chapter 7.4; only pairs where both the source and destination target are in their respective block's candidate set can receive a finite cost. All other entries, as well as structurally disallowed transitions, receive $+\infty$. The matrix has no symmetry requirement; diagonal entries are fixed at 0.

Not every target pair admits a transition. PostgreSQL is the most restricted backend: it is outgoing-only, as shown in Table 4. PostgreSQL serves as the seeding data source, the authoritative store that determines which vertices are alive at any given temporal interval. Consequently, every query must access it. Any incoming transition would duplicate this access, as the query would first need to access the database to seed the available vertices, yield to another backend for intermediate computation, and then re-enter PostgreSQL. Re-entry is technically possible through VALUES clauses, but no case has been identified where it would reduce total cost; the restriction is therefore enforced unconditionally. This constraint is architectural, not fundamental to the cost model: a system with a different authoritative backend, such as an embedded key-value store, could lift it. Separately, PostgreSQL is excluded from looping regions, because SQL's declarative execution model cannot express iteration; loops require stored procedures, which would turn a query into a mutation, as further elaborated in Chapter 7.2.1.

The remaining backends face no directional restrictions. The embedding backend permits all transitions; because no underlying store exists to validate against, restricting transitions would encode premature assumptions. The interpreter accepts incoming transitions from every backend, which makes it the universal sink in the transition graph. This property is required for it to serve as the universal fallback and is the pairwise counterpart to the feasibility guarantee established in Chapter 7.2.

Goto

	I	P	E
I	0	×	c
P	c	0	c
E	c	×	0

SwitchInt

	I	P	E
I	0	×	c
P	c	0	c
E	c	×	0

GraphRead

	I	P	E
I	0	×	×
P	×	×	×
E	×	×	×

I: Interpreter. P: PostgreSQL. E: Embedding.
Rows: source. Columns: destination. c: transfer cost. ×: disallowed.

Table 4: Transition matrices by terminator kind.

The terminator kind of the source block further restricts which transitions an edge permits, because the terminator determines how control is transferred. `Goto` and `SwitchInt` edges permit any transition between mutually supported targets. This property is required by the block splitting pass in Chapter 7.4, which assumes that a zero-argument `Goto` between split fragments always admits a cross-backend transition. `GraphRead` edges allow only interpreter-to-interpreter transitions, as graph effects require runtime coordination that no compiled backend can provide.

Each cross-backend transition cost has two components. The first is a fixed overhead independent of data volume, shown in Table 5. This overhead reflects the cost of yielding to the runtime for serialization and deserialization. Without it, the solver would be free to switch backends whenever the data-proportional component is zero, producing spurious transitions between blocks that carry no live data. The fixed overhead prevents this. Transitions to the interpreter are cheaper than transitions between other backends, because the interpreter is co-located with the runtime and requires no additional dispatch.

	I	P	E
I	0	×	4
P	8	0	12
E	4	×	0

I: Interpreter. P: PostgreSQL. E: Embedding.
Rows: source. Columns: destination. ×: disallowed.

Table 5: Fixed backend switch overhead per target pair.

The second component is proportional to the volume of live data that must cross the edge. Traversal-aware liveness analysis (Chapter 6.2) determines which locals are live at the successor's entry. Block arguments are included in this set, as they represent values that must be passed across the edge. The transfer cost is the sum of the estimated sizes of all live locals in information units. Chapter 7.3.1 details how these estimates are derived. The total edge cost, fixed overhead plus data-proportional transfer, populates each entry of the transition matrix t_e.

7.3.1. Size Estimation

The data-proportional component of a backend transition requires knowing how much data each live value carries. A naive measure, counting the number of live locals, fails to reflect actual data volume: a local holding a five-element tuple is strictly larger than a local holding a single integer. The ideal measure would be the exact byte size, but the language supports arbitrary-precision integers and dynamically sized strings, so

byte-level sizes are not statically available. Information units provide an intermediate granularity: one information unit corresponds to one primitive value, and compositions of primitives yield a proportional count. This abstraction is coarser than bytes but fine enough to distinguish a scalar from a compound structure.

To increase precision for dynamic containers, the model introduces a second axis: cardinality. Every value carries both a per-element information content and a cardinality. For non-container values the cardinality is one; for collections it reflects the estimated number of elements. The two axes are independent, which allows operations over collections to be tracked without loss of precision: removing an item from a collection decreases the cardinality, whereas projecting to an inner field decreases the information content.

Because values flow through join points and unions, neither dimension can be modeled as a scalar. A union could resolve to any of its members, each with a different size; a join point merges values from branches that may carry different cardinalities. Both dimensions are therefore represented as ranges with a lower and upper bound. Unbounded upper limits are permitted for types whose maximum size cannot be determined, such as lists of unknown length. The estimated transfer size of a value is computed by first forming the product range and then taking its midpoint. Let $r = [l, u]$ denote a range with inclusive lower bound l and inclusive upper bound u, let $\text{units}(v) = [l_u, u_u]$, and let $\text{cardinality}(v) = [l_c, u_c]$. Then:

$$\begin{aligned} \text{midpoint}([l, u]) &= \frac{l + u}{2} \\ \text{size}(v) &= \text{midpoint}([l_u * l_c, u_u * u_c]) \end{aligned} \tag{12}$$

The analysis operates in two phases.

Static Phase. Because the MIR retains type information from the HIR and body signatures are already monomorphized, the static phase can derive information units directly from the type structure. For any type that does not involve dynamic collections, this produces an exact range without dataflow analysis. The static phase also seeds the initial state of the dynamic phase, which reduces the number of iterations required to reach a fixpoint. The static rules are:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

$$[l_1, u_1] \sqcap [l_2, u_2] = \begin{cases} [\max(l_1, l_2), \min(u_1, u_2)] & \text{if } \max(l_1, l_2) \leq \min(u_1, u_2) \\ [0, 0] & \text{otherwise} \end{cases}$$

$$\text{units}(T) = \begin{cases} [1, 1] & \text{if } T \in \{\text{Primitive, Closure}\} \\ \sum_{f \in T} \text{units}(f) & \text{if } T \in \{\text{closed Struct, Tuple}\} \\ \bigsqcup_{i \in 1..n} \text{units}(T_i) & \text{if } T = T_1 \cup \ldots \cup T_n \\ \bigsqcap_{i \in 1..n} \text{units}(T_i) & \text{if } T = T_1 \cap \ldots \cap T_n \\ [0, 0] & \text{if } T \in \{\bot, \text{Param, Infer}\} \\ \diamondsuit & \text{if } T \in \{\top, \text{List<T>, Dict<K, V>, open Struct}\} \end{cases} \tag{13}$$

Primitives and closures each carry one information unit. A primitive is an indivisible atom in the type system; a closure is a function pointer, and while it captures an environment, the pointer itself is a single scalar reference. Closed structs and tuples

sum the units of their fields. Unions widen the range across all members; intersections tighten it to their overlap, degenerating to the empty range when no overlap exists. The uninhabited never type (⊥) carries zero units because no value of that type can be constructed. Type variables (`Param`, `Infer`) also carry zero units, but for a different reason: they are eliminated during type simplification and do not survive to the MIR.

Dynamic containers, open structs, and unknown (`T`) types are deferred to the dynamic phase (⟐). Their size depends on runtime cardinality, which the static phase cannot determine. Open structs are deferred because only a subset of their fields is known; costing the known subset would underestimate. Lists and dictionaries are covariant over their element type, so deferral also allows the dynamic phase to narrow the element type at usage sites.

Dynamic Phase. Types marked ⟐ by the static phase cannot be resolved from type structure alone. The dynamic phase resolves them by propagating sizes through assignments via forward dataflow over a saturating join semilattice. Because sizes grow monotonically and the upper bound of the ranges is large, convergence to a true fixpoint is guaranteed but prohibitively slow in degenerate cases, such as loops that push an element on every iteration. Instead, a fixed iteration bound caps both the per-body dataflow and the per-SCC outer pass. Because the analysis starts from bottom and grows, early termination yields an underestimate rather than an upper bound. This is a deliberate design choice: underestimation is preferred over unbounded iteration, and transfer costs derived from a smaller estimate are never unsafe, only potentially suboptimal.

Each assignment's size depends on its operands, which the analysis can resolve from the body, with one exception: function arguments. Their sizes are supplied by the caller and cannot be derived from the callee alone. If the argument type was statically resolved, the constant range suffices. For dynamic arguments, the dependency must be tracked explicitly. Rather than introducing a separate representation, the model extends the scalars used in ranges to affine equations over the function's argument sizes. Constants are the degenerate case where all coefficients are zero. The result of both phases is a footprint for each local: the pair of its information range and its cardinality, where each scalar is:

$$y = c_1 * a_1 + c_2 * a_2 + \ldots + k \tag{14}$$

where c_i tracks argument a_i's contribution and k is the argument-independent constant. The return footprint, determined by joining all return points in the body, may itself be an affine equation. Function application follows naturally from the affine representation: at each call site, the caller substitutes its own argument footprints into the callee's return equation. The result may again be an affine equation if the caller's arguments are themselves unresolved.

Because function calls create inter-body dependencies, size estimation is a global analysis pass. The call graph introduced in Chapter 6.3.1 is used and condensed into strongly connected components. Similarly to inlining, we process callees before callers to ensure that the footprint is available during dynamic analysis. Mutually recursive functions form strongly connected components where no ordering of members exists. Within each component, the analysis iterates over all member bodies and recalculates

their sizes until either a fixpoint or the fixed iteration bound is reached. Stabilization is more likely than the bound suggests: in practice, at least one member of a recursive chain has a return type that does not depend on its inputs, which anchors the remaining estimates. At the true fixpoint, the order in which members are visited within a component does not affect the result. Under the iteration cap, visitation order can influence the final estimate, but the monotonicity of the join guarantees that any order produces a sound underestimate.

The path premium shown in Equation 10 is also derived from this analysis, as part of the block cost assembly elaborated in Chapter 7.4. Each vertex path maps to a specific storage location whose size is derived from its physical representation. Depending on the data contained, the size is either static or dependent on the type of the vertex currently processed.

7.4. Block Cost Assembly

The solver assigns exactly one execution target to each basic block. Statement placement, however, operates at instruction granularity: each statement and terminator receives its own supported-target set. A block whose statements disagree on target support cannot receive a coherent assignment. The naive resolution is to intersect all per-statement target sets within a block and assign the block to the result. This approach has two flaws. First, it demotes blocks to the common denominator: a single interpreter-only statement, such as a function call, forces the entire block onto the interpreter regardless of how many statements are PostgreSQL-native. Second, the intersection may be empty when no single backend supports every statement in the block, which would make a universal fallback a hard architectural requirement rather than a design choice.

Block splitting resolves this by subdividing each block into contiguous regions of uniform target support, as described in Algorithm 12. The pass walks each block's statements and tracks the supported-target set derived from statement placement. When the set changes between consecutive statements, a new region begins. Once all statements are resolved, the terminator is checked: if its target set is a superset of the last region's, the terminator joins that region. If it narrows the set, the terminator receives its own region. A superset is tolerated because the cost of an additional backend transition outweighs the benefit of a wider target domain for a single terminator. Each region becomes its own basic block, connected to the next by a zero-argument `Goto` terminator that preserves the original control flow. Zero-argument `Goto` transitions must therefore be supported by every backend: without this guarantee, the splitting invariant cannot hold. To avoid inflating costs through these structural subdivisions, each inserted `Goto` is costed at zero. Transition costs between split blocks are not discounted and are still computed at the regular rate described in Chapter 7.3.

Algorithm 12: Basic block splitting

```
1: function Split(B, costs)
2:     for each block b ∈ B do
3:         if b has no statements then
```

```
        b remains one region with target set targets(terminator(b))
        continue
    end

    regions ← empty list
    current ← targets(s_0)
    start ← 0

    for each statement s_i in b do
        if targets(s_i) ≠ current then
            append [start, i) with target set current to regions
            start ← i
            current ← targets(s_i)
        end
    end
    append [start, |b|) with target set current to regions

    ▷ Separate terminator if it narrows support
    T ← targets(terminator(b))
    if T ⊉ current then
        append terminator region with target set T to regions
    end

    ▷ Replace b with |regions| blocks chained by zero-argument Goto
    for r_j, r_{j+1} in regions do
        terminator(r_j) ← Goto(r_{j+1})
    end
    terminator(r_{|regions|}) inherits terminator(b)
  end

  remap block references and cost vectors to new layout
end
```

Transition costs are computed after splitting, because the new block layout introduces edges that did not exist in the original CFG. The transition matrices described in Chapter 7.3 are therefore evaluated on the split layout, where each edge receives a cost matrix reflecting liveness-based transfer costs and directionality constraints. With both statement costs and transition costs in place, the per-block cost $c(\mathtt{bb}_n, x_{\mathtt{bb}_n})$ from Equation 10 can be assembled for every candidate target of each block. The base cost is the sum of per-statement costs and the terminator cost. Both are guaranteed to be finite and non-negative for every candidate target, because splitting ensures that all instructions in the same block share the same supported-target set.

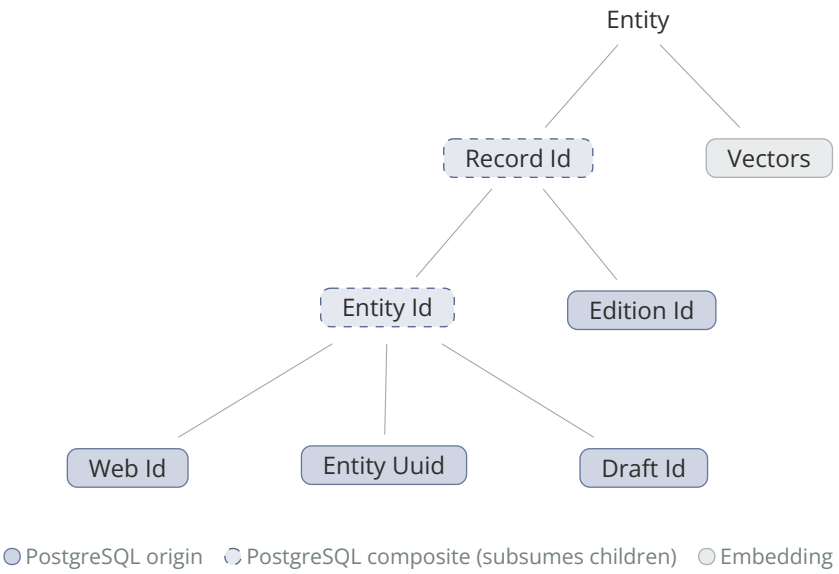


Figure 13: Vertex path hierarchy (subset). Composites share their children's origin.

The path transfer premium from Equation 10 is then added to the base cost. The premium is charged once per block: a traversal analysis pass walks each block's statements and records which vertex paths are accessed. Vertex paths form a hierarchy in which composite paths subsume their children. If a composite path such as `RecordId` is accessed, its constituent paths (`EntityId`, `EditionId`, and their children) do not contribute additional cost, because the composite already covers them. Subsumption is downward only: if all children of a composite are accessed individually, they are not promoted to the composite, because the composite may include additional data beyond its children and promotion would inflate the cost. For each recorded path whose origin backend differs from the candidate target, the estimated transfer size from Chapter 7.3.1 is charged as a premium. Vertex accesses that do not resolve to a concrete path are treated conservatively as full hydration, charging the entire path set. When the candidate target is the natural origin for every accessed path, the premium is zero. With per-block costs and per-edge transition matrices assembled, the solver receives the two structures that instantiate Equation 11: the unary term $c(\mathtt{bb}_n, x_{\mathtt{bb}_n})$ for each block and candidate target, and the pairwise term $t_e(x_{\mathrm{src}(e)}, x_{\mathrm{dst}(e)})$ for each edge and target pair.

7.5. Solving

Exact global optimization of Equation 11 is intractable (Chapter 7.1). Because the cost model decomposes into unary and pairwise terms, we adopt a greedy approach with a value-ordering heuristic. Tomas Karnagel, et al. [37] demonstrate that greedy local minimization over operator placements on heterogeneous hardware achieves near-optimal results without exhaustive search.

Domain pruning reformulates the placement problem as a constraint satisfaction problem (CSP) and eliminates infeasible target assignments before the solver runs (Chapter 7.5.1). The reduced branching factor accelerates every subsequent decision without affecting correctness.

Once domains are pruned, the CFG is condensed into a DAG of placement regions to establish a topological ordering. Each SCC becomes a placement region: trivial components become trivial placement regions; multi-block components become cyclic regions. While greedy assignment is sufficient for trivial regions, cyclic regions require a more exhaustive strategy. Any transition inside a loop compounds with every iteration; the solver therefore applies branch-and-bound to small cyclic components instead of greedy assignment.

To rank a candidate target $\Omega \in D(\mathtt{bb}_n)$ during greedy placement, the solver uses the estimate $\hat{c}(\mathtt{bb}_n, \Omega)$. This is a value-ordering heuristic, not an exact evaluation of Equation 11: the true edge terms depend on both endpoints and cannot be evaluated before all blocks are assigned. The estimate starts with the per-block cost $c(\mathtt{bb}_n, \Omega)$ from Equation 10 and adds transition charges for every non-self-loop edge incident to $\mathtt{bb}_n$.

For an assigned neighbor, the realized transition cost is charged directly. For an unassigned neighbor, the heuristic performs a local lookahead: given the current candidate Ω and the edge to the neighbor, it determines the target the neighbor would choose to minimize its own block cost plus the transition cost, then charges only the transition component. The neighbor's block cost steers the selection but is not attributed to $\mathtt{bb}_n$, because $E(x)$ charges it to the neighbor. This lookahead is the primary heuristic element: it lacks global context but permits local reasoning over the immediate subgraph.

Formally, let $A \subseteq B$ be the set of already-assigned blocks and $x_{\mathtt{bb}_n}$ denote the current target of $\mathtt{bb}_n \in A$. The sets E^- and E^+ collect all non-self-loop edges terminating in and originating from $\mathtt{bb}_n$:

$$\begin{aligned} E^-(\mathtt{bb}_n) &= \{e \in E \mid \mathrm{dst}(e) = \mathtt{bb}_n \wedge \mathrm{src}(e) \neq \mathtt{bb}_n\} \\ E^+(\mathtt{bb}_n) &= \{e \in E \mid \mathrm{src}(e) = \mathtt{bb}_n \wedge \mathrm{dst}(e) \neq \mathtt{bb}_n\} \end{aligned} \tag{15}$$

The estimate $\hat{c}$ combines the per-block cost with weighted transition charges over both edge sets:

$$\hat{c}(\mathtt{bb}_n, \Omega) = c(\mathtt{bb}_n, \Omega) + \sum_{e \in E^-(\mathtt{bb}_n)} w_e \tau_e^-(\mathtt{bb}_n, \Omega) + \sum_{e \in E^+(\mathtt{bb}_n)} w_e \tau_e^+(\mathtt{bb}_n, \Omega) \tag{16}$$

The transition charges τ^- and τ^+ distinguish assigned from unassigned neighbors. For assigned neighbors, the realized cost is used directly. For unassigned neighbors, a lookahead δ selects the target that minimizes the neighbor's block cost plus the transition:

$$\begin{aligned} \tau_e^-(\mathtt{bb}_n, \Omega) &= \begin{cases} t_e\big(x_{\mathrm{src}(e)}, \Omega\big) & \text{if } \mathrm{src}(e) \in A \\ t_e(\delta_e^-(\mathtt{bb}_n, \Omega), \Omega) & \text{otherwise} \end{cases} \\ \delta_e^-(\mathtt{bb}_n, \Omega) &\in \arg\min_{\delta \in D(\mathrm{src}(e))} (c(\mathrm{src}(e), \delta) + t_e(\delta, \Omega)) \\ \tau_e^+(\mathtt{bb}_n, \Omega) &= \begin{cases} t_e\big(\Omega, x_{\mathrm{dst}(e)}\big) & \text{if } \mathrm{dst}(e) \in A \\ t_e(\Omega, \delta_e^+(\mathtt{bb}_n, \Omega)) & \text{otherwise} \end{cases} \\ \delta_e^+(\mathtt{bb}_n, \Omega) &\in \arg\min_{\delta \in D(\mathrm{dst}(e))} (c(\mathrm{dst}(e), \delta) + t_e(\Omega, \delta)) \end{aligned} \tag{17}$$

The boundary weight w_e dampens cross-region transitions during the forward pass to incentivize intra-component coherence:

$$w_e = \begin{cases} 1 \text{ if } e \text{ is intra-region} \\ \frac{1}{2} \text{ if } e \text{ is a cross-region boundary edge of a cyclic region} \end{cases} \tag{18}$$

Any transition inside a loop is replayed on every iteration and therefore accumulates cost that Equation 11 does not model; the halved weight compensates for this. During adjustment passes, w_e is set to 1 for all edges; with all neighbors assigned, $\hat{c}(\texttt{bb}_n, \Omega)$ then collects precisely the terms of $E(x)$ that depend on $\texttt{bb}_n$'s assignment, which is the exact local conditional. Besag's convergence guarantee therefore applies: each accepted update strictly decreases $E(x)$, and the alternating sweeps terminate at a coordinatewise local minimum. The sums range over edges, not adjacent blocks, so parallel edges from a `SwitchInt` targeting the same successor contribute independently; in the absence of frequency data, this multiplicity is a natural proxy for how strongly two blocks are coupled [10].

Cyclic placement regions can be solved greedily, but solution quality degrades: no topological order exists within the component, so no block has an anchor from which to propagate a reliable estimate. For small strongly connected components, the solver replaces greedy assignment with branch-and-bound, which searches the space exhaustively to increase intra-region coherence. The search selects the next block to assign using the minimum remaining values heuristic: the block with the fewest candidates in its pruned domain is assigned first, because it has the highest chance of causing a failure and the fewest alternatives to explore. After each assignment, domains of all unassigned blocks in the component are narrowed in both directions: successors are restricted to targets reachable from the assigned target, predecessors to targets from which the assigned target is reachable. A domain emptied by this narrowing proves the partial assignment has no feasible completion; the search backtracks immediately. Pruning uses a lower bound on the cost of completing the current partial assignment: for each unassigned block, the bound independently minimizes block cost and transition cost over the remaining domain. A branch is abandoned when the accumulated cost plus this bound exceeds the worst of the retained solutions. The search retains the three lowest-cost complete assignments rather than only the best: this provides alternatives for the rewind mechanism and hedges against the heuristic's inexactness, since the true cost under full boundary context may reorder the ranking. Larger components fall back to greedy assignment using the same minimum remaining values ordering; without exhaustive search, these assignments rely more heavily on the adjustment passes described below.

The forward pass processes placement regions in topological order over the condensation DAG. For each trivial region, $\hat{c}$ ranks all candidate targets; the cheapest is selected and the alternatives are retained in sorted order. For each cyclic region, branch-and-bound assigns all member blocks simultaneously.

A greedy forward pass may commit to an assignment that leaves no feasible option for a downstream region. Domain pruning reduces this risk but does not eliminate it. When a region cannot be assigned, the solver backtracks along the topological order. Trivial and cyclic regions differ in how they provide alternatives. Trivial regions retain a sorted list of candidates from $\hat{c}$; on backtrack, the next-cheapest candidate is selected. Cyclic regions retain the top three branch-and-bound solutions; on backtrack, the next-

ranked solution replaces the current one. If all retained solutions for a cyclic region are exhausted, the solver enters a perturbation fallback: it identifies the block within the cycle whose next-best candidate has the smallest cost delta, replaces that block's assignment, and replays all subsequent blocks in the cycle using greedy selection. This is repeated until an alternative is found or all candidates are exhausted. A feasible assignment always exists: the interpreter is present in every domain and interpreter-to-interpreter transitions are universally valid, so an all-interpreter labeling is always feasible.

The forward pass assigns with partial information: at the time a region is processed, its successors are unassigned, and $\hat{c}$ substitutes optimistic estimates for their contributions. Adjustment passes correct this asymmetry by re-evaluating each region with the exact local conditional of $E(x)$, as described above. For trivial regions, this is a single-block update that minimizes $E(x)$ directly. For cyclic regions, the solver re-runs the full region optimization; the new assignment is accepted only if the total regional cost under $E(x)$ is strictly lower than the incumbent. The first adjustment pass sweeps in reverse topological order; if any assignment changes, a subsequent pass sweeps in the forward direction, and so on, alternating until a full pass produces no change.

The separable cost model from Equation 11 was chosen in part to enable PBQP as a future preprocessing step. PBQP admits R1 and R2 reductions that fold degree-1 and degree-2 nodes into their neighbors' cost vectors without information loss; empirically, these reductions eliminate most nodes before any heuristic runs. Any irreducible remainder still requires a heuristic fallback; the greedy framework described above is therefore the foundation: PBQP reductions shrink the graph the solver receives, while the greedy solver handles the irreducible core. The integration is deferred to future work (Chapter 10.1.1.3).

7.5.1. Domain Pruning

Beyond energy minimization, the placement problem from Chapter 7.1 exposes the structure of a CSP. Each basic block is a variable with domain $D(\texttt{bb}_n) \subseteq T$: the supported-target set from statement placement on the split layout. Each CFG edge imposes a binary constraint on the target pairs of its endpoints through its transition matrix. Because the CFG is a directed multigraph, a pair of adjacent blocks may be connected by multiple edges in either direction, each carrying its own matrix. The aggregate constraint between two blocks composes all such edges:

$$\begin{aligned} R_e(a,b) &\equiv t_e(a,b) < \infty \\ E_{xy} &= \{e \in E : \mathrm{src}(e) = x, \mathrm{dst}(e) = y\} \\ C_{xy}(a,b) &\equiv \big(\forall e \in E_{xy} : R_e(a,b)\big) \\ &\quad \wedge \big(\forall e \in E_{yx} : R_e(b,a)\big) \end{aligned} \tag{19}$$

R_e extracts feasibility from the transition cost t_e of Equation 11, disregarding the magnitude: the CSP concerns itself with which transitions are permitted, not with their relative expense. E_{xy} collects the directed edges from x to y. The aggregate constraint C_{xy} requires a candidate pair to satisfy every directed edge between the two blocks simultaneously, because parallel edges from a `SwitchInt` terminator each impose an

independent constraint. The converse orientation $R_e(b, a)$ for edges in E_{yx} preserves the asymmetry of the transition matrix: an edge from y to x is indexed by (t_y, t_x), so determining whether $a \in D(x)$ is compatible with $b \in D(y)$ requires $R_e(b, a)$, not $R_e(a, b)$.

A value $a \in D(x)$ is supported with respect to neighbor y when at least one value in the neighbor's domain satisfies the aggregate constraint:

$$\text{supported}(a, x, y) \equiv \exists b \in D(y) : C_{xy}(a, b) \tag{20}$$

The CSP structure and the supportedness predicate together admit standard domain pruning. AC-3, a worklist algorithm that enforces arc consistency by iteratively removing values from domains that lack a support witness in any neighboring domain, removes unsupported values from all domains before the solver runs. The mechanism is natural from the perspective of edges: processing the edge between x and y may narrow $D(x)$ by removing unsupported values; all other edges incident to x must then be re-examined, because neighbors at their far ends may have relied on the removed value as their sole support witness [55].

AC-3 is sound over any binary CSP with finite domains: it never removes a value that belongs to a valid complete assignment. Because C_{xy} is a binary relation over finite domains, the standard guarantees transfer directly. The interpreter's role as universal fallback (Chapter 7.2, Chapter 7.3) guarantees that no domain can empty. Even without this guarantee, an empty domain would not be a correctness failure: it would indicate that no valid assignment exists, a fact the solver would eventually reach independently. After the fixpoint, the transition matrices are pruned to match the narrowed domains; the solver operates on the reduced CSP without re-checking eliminated candidates.

7.6. Block Fusion

Once solving concludes, every basic block carries a concrete target assignment $x_{\texttt{bb}_n}$ ($\forall \texttt{bb}_n \in B : \texttt{bb}_n \in A$). Block fusion partially reverses the splitting from Chapter 7.4. The solver requires a uniform target domain $D(\texttt{bb}_n)$ for every block, which necessitates splitting to maximal granularity without over-constraining any single domain. Each split point is a potential backend transition that the solver may or may not take. Because splitting fragments the body significantly, block fusion merges consecutive same-target blocks where $x_{\texttt{bb}_i} = x_{\texttt{bb}_j}$, reducing CFG edges and therefore internal control overhead. The pass is deliberately narrow: it fuses only consecutive blocks that exhibit the structural characteristics introduced by splitting, so as not to break SSA invariants or introduce performance regressions. This restriction is safe, because any naturally fusable block pairs in the original body were already reduced by CFG simplification (Chapter 6.3.2).

Algorithm 13 outlines the procedure. Fusability requires a single predecessor connected by an argument-free `Goto`, no block parameters, and identical target assignments, which are the conditions that splitting introduced, in addition to the fixed target. Head resolution collapses transitive chains, so a sequence $\texttt{bb}_i \to \texttt{bb}_j \to \texttt{bb}_k$ where all three share a target fuses into $\texttt{bb}_i$. After merging, the fused blocks leave gaps in the identifier space. Once completed, the basic blocks are remapped to restore a dense identifier space, improving space-time performance of subsequent passes. The fused

layout is the input to island construction: each remaining cross-target edge becomes an island boundary.

Algorithm 13: Basic block fusion

```
1:  function Fuse(B, targets)
2:      ▷ Phase 1: head resolution
3:      for each block b ∈ B do
4:          head(b) ← b
5:      end
6:      for each block b ∈ B in reverse postorder do
7:          if Fusable(b) then
8:              p ← the predecessor of b
9:              head(b) ← head(p)
10:         end
11:     end
12:
13:     ▷ Phase 2: statement merging
14:     for each block b ∈ B in reverse postorder where head(b) ≠ b do
15:         append statements of b to head(b)
16:         terminator(head(b)) ← terminator(b)
17:     end
18:
19:     ▷ Phase 3: compaction
20:     assign contiguous identifiers to blocks where head(b) = b
21:     rewrite all block references and truncate
22: end
23:
24: function Fusable(b)
25:     return b ≠ bb_0
            ∧ b has exactly one predecessor
            ∧ terminator(predecessor(b)) = Goto(b)
            ∧ the Goto carries no arguments and b has no parameters
            ∧ targets(predecessor(b)) = targets(b)
26: end
```

7.7. Islands

After solving and fusion, the CFG contains contiguous regions of blocks that share the same target. These regions are the natural units of execution scheduling: each is dispatched to its assigned backend as one unit by the runtime (Chapter 8). Following Tomas Karnagel, et al. [37], we call them execution islands. Formally, an island is a maximal connected component of same-target blocks in the CFG:

$$
\begin{aligned}
E_{=} &= \left\{e \in E : x_{\mathrm{src}(e)} = x_{\mathrm{dst}(e)}\right\} \\
\mathcal{I} &\coloneqq \text{weakly connected components of } (B, E_{=}) \\
&= \langle I_1, \ldots, I_m \rangle \\
\\
I_n &\subseteq B \\
B &= \bigcup_{i \in \mathcal{I}} i \\
&\quad \forall n, m \in 0..|\mathcal{I}|. n \neq m \implies I_n \cap I_m = \varnothing \\
\\
\mathrm{target}(I_n) &: \mathcal{I} \to T \\
&\coloneqq \text{target of } I_n \\
\\
\mathcal{I}_\Omega &= \{I_n \in \mathcal{I} : \mathrm{target}(I_n) = \Omega\} \\
\Omega_i &\coloneqq i\text{-th element of } \mathcal{I}_\Omega
\end{aligned}
\tag{21}
$$

$E_{=}$ restricts the CFG to edges whose endpoints share the same assignment; each connected component of $(B, E_{=})$ defines a single execution island. The partition property guarantees that every block belongs to exactly one island: no block is orphaned and no block appears in two islands. The blocks of each island share the same target by construction, so $\mathrm{target}(I_n)$ is well-defined.

Execution scheduling alone does not require path tracking, but the runtime must also retrieve vertex data: an island assigned to a target that is not the origin for a path it accesses cannot execute without that data being fetched first. Each execution island therefore tracks two path sets. The requires set contains the paths the island needs from external providers:

$$
\mathrm{requires}(I_n) = \{p \in \mathrm{paths}(\mathtt{bb}_k) : \mathtt{bb}_k \in I_n, \mathrm{origin}(p) \neq \mathrm{target}(I_n)\} \tag{22}
$$

The provides set records which paths the island makes available to downstream consumers. It starts empty and grows on demand: when a downstream island requires a path, the nearest ancestor on the path's origin target is located. If that ancestor, or any of its same-target ancestors via the inheritance chain, already provides the path, no new registration is needed. Otherwise, the path is added to the nearest ancestor's provides set. Only paths that are actually needed are ever registered, so the runtime never issues a retrieval that no island consumes.

To determine both the execution order and the data dependencies, a condensed multigraph is constructed over $\mathcal{I}$ with three edge kinds:

- **Control-flow.** For each cross-target edge in the CFG, a control-flow edge is inserted between the source and destination islands if one does not already exist: $E_{\mathrm{cf}} = \left\{(I_i, I_j) \mid e \in E \setminus E_{=}, \mathrm{src}(e) \in I_i, \mathrm{dst}(e) \in I_j\right\}$. These edges encode island-level sequencing. Transitions in loops do not guarantee that the island graph is a DAG.
- **Inheritance.** For each island I_n, the nearest strict dominator I_m with $\mathrm{target}(I_m) = \mathrm{target}(I_n)$ receives an inheritance edge. Dominance is computed over the control-flow island graph. The runtime retains fetched data across same-target dispatch, so every path in I_m's provides set is transitively available to I_n. Because I_m dominates I_n, it is resolved first, and its provides set is fully populated before I_n is processed.

- **Data-flow.** Each path in $\text{requires}(I_n)$ must be resolved against a provider. For each such path, the nearest strict dominator assigned to the path's origin target is located. A data-flow edge connects this dominator to I_n. If the path is already in the dominator's provides set, inherited from its own same-target ancestors, no new registration is needed. Otherwise, the path is added to the dominator's provides set.

Control-flow and inheritance edges are resolved first, because the inheritance step populates provides sets that downstream consumers of I_n depend on. Data-flow edges are resolved second, against the already-populated provides sets. In some cases, no dominator on any origin target exists for a required path. A synthetic data island is inserted to satisfy the dependency: a node with no member blocks, assigned to the path's origin target, whose sole purpose is to issue the fetch. Data islands have no requires set and execute before any of their consumers. The cost of the fetch is already captured by the per-block path transfer premium from Equation 10, which charges every non-origin path access regardless of whether the provider is an execution island or a synthetic data island. The solver's cost minimization naturally colocates computation with data sources, so synthetic data islands are rare in practice. The resulting island dependency graph, together with the per-block target assignments, is the final output of the placement pipeline and the input to Chapter 8.

8. Code Generation and Execution

Where compilers such as Rust or Swift lower their intermediate representation through LLVM to a single machine target, the HashQL pipeline must lower its MIR to multiple heterogeneous execution backends simultaneously. The task is more constrained: each backend can express only a subset of the MIR, and when a body contains islands on different execution targets, the runtime must coordinate between them. The code generation and execution pipeline acts on the island dependency graph and per-block target assignments produced by the placement pipeline (Chapter 7). Islands are compiled into backend-specific artifacts and subsequently dispatched at runtime through the orchestrator. We implement two backends end-to-end: PostgreSQL, which compiles MIR fragments into SQL, and the interpreter, which tree-walks the MIR directly. A third target, the embedding backend, participates in placement analysis but is not implemented beyond that phase: the HASH graph does not yet provide a vector store. The execution infrastructure is backend-agnostic; adding a new target requires a code-generation module and cost entries, not changes to the orchestrator or the island model [26, 48, 56].

The pipeline operates in two phases, following the same separation as traditional compilers. During code generation, each backend compiles its assigned islands into backend-specific artifacts. The PostgreSQL backend compiles across all graph effect operations into a single prepared statement with its required parameters; the interpreter produces no separate artifact and operates directly on existing MIR bodies. During execution, the runtime dispatches these artifacts according to the island dependency graph without re-running the compilation pipeline [3].

The runtime consists of the orchestrator and the interpreter. The orchestrator walks the island dependency graph, dispatching each island to its assigned backend and coordinating transitions when control flow crosses target boundaries. The interpreter drives program execution; at effectful terminators it yields a suspension that the orchestrator fulfills before resuming. This suspension protocol follows the coroutine model: an effectful terminator pauses execution, and the orchestrator resumes it once the effect has been fulfilled.

Execution proceeds sequentially over each body in the pipeline. While some backends, such as PostgreSQL, batch across graph effect pipelines by compiling all their islands into a single prepared statement, the orchestrator itself does not batch at the pipeline level; it reuses data that prior pipelines have already provided. The orchestrator man-

ages the state required by the provides and requires contract described in Chapter 7.7, mediating every inter-target data transfer.

8.1. Interpreter

The interpreter follows a tree-walking design and executes the MIR directly without a prior compilation step. We chose a tree-walking interpreter to validate the feasibility of the heterogeneous execution framework first; a register-based bytecode VM is discussed as future work in Chapter 10.1.1.2.

The interpreter is stack-based: each function call creates a new frame that stores the locals, the current program counter, and a reference to the executing body. Because each body declares its locals statically (Chapter 6), storage is pre-allocated, mirroring the stack behaviour of traditional compiled languages.

Execution follows a loop in which each step advances the program counter to the next statement. When the interpreter encounters a terminator, it dispatches control flow: branches advance the program counter to another basic block, return pops the current frame, and effectful terminators suspend execution. Function application is modelled as a statement rather than a terminator (Chapter 6.1), so a new stack frame is pushed and executed inline. Once the call has completed, the return value is written to the specified location. This is safe because function application has a single continuation and no alternate control-flow paths (Chapter 6.1). Two extensions suspend execution when coordination with the orchestrator is required.

The first extension is coroutine suspension at effectful terminators: the execution of an effect requires handling outside the interpreter, so control must yield back to the runtime. Catastrophic failure, such as a network or database outage, may prevent the runtime from fulfilling the request, in which case computation is not resumed. This failure is outside the language semantics: the language itself does not expose a failure path to the program. Referential transparency ensures that pure computation performed before the suspension requires no cleanup. The current work implements only read operations; future write operations would benefit from rollback semantics to preserve this guarantee under catastrophic failure. Block parameters make the fulfillment of a coroutine straightforward: the returned value is set as the block argument, and the program pointer is set to the entry of the target block [11:5.2].

The second extension is island-aware suspension. While the interpreter can evaluate any statement, the placement pipeline may assign blocks within the same body to different execution targets (Chapter 7). Because only graph-effect bodies undergo execution analysis and receive island assignments (Chapter 7), this check applies only to the root stack frame; callees reached through function application do not carry island assignments. At each block boundary, the interpreter checks whether the transition would cross an island boundary; if so, execution yields back to the runtime. Traversal-aware dataflow analysis (Chapter 6.2) is run during code generation to determine which locals and vertex paths must be transferred at each island boundary, satisfying the cost model described in Chapter 7.1. When execution later resumes at an interpreter

island, values produced by the prior island are written into the expected locals, and the program pointer is set to the entry block of the island.

8.2. PostgreSQL

The PostgreSQL backend translates a subset of the MIR into declarative SQL. Not all MIR constructs have SQL equivalents: closures, function application, and nested graph reads are rejected at placement time (Chapter 7.2.1). The translation requires three structural preconditions, all guaranteed by the placement pipeline (Chapter 7.2.1): each PostgreSQL island is acyclic (loops are rejected), has a single entry block, and contains only side-effect-free instructions. Combined with the SSA property of the MIR, no mutable state exists and every local is assigned exactly once on each control-flow path, which makes conversion into a SQL expression feasible. The operation is conceptually the reverse of ANF normalization (Chapter 5.1): where that pass decomposes expression trees into a sequence of atomic assignments, the PostgreSQL backend folds assignments back into nested sub-expression trees.

SQL lacks let-bindings: there is no way to name an intermediate result within an expression and reference it later. The compiler therefore converts the DAG into a tree by duplicating blocks below each join point and compiles each control-flow path independently, a form of if-conversion that maps control dependence to data dependence as outlined by J. R. Allen, et al. [4]. The overhead of duplication is purely textual: conditional expressions short-circuit, so PostgreSQL evaluates only the taken arm per row, and prepared statements amortize the increased plan time by reusing the plan across invocations. Because the result is a scalar expression rather than a subquery, the query planner retains full visibility and predicate pushdown and constant folding apply without barrier.

An alternative would be to deduplicate at join points using CTE subqueries. The construction requires knowing which predicate controls each block parameter. While this can be recovered from SSA and CFG analysis, gated single assignment (GSA) provides a principled basis: it replaces block parameters (φ-functions) with γ-functions that carry the controlling predicate, converting each join point into a merge point. The point of view inverts: rather than the branch selecting an entire path, the merge selects a value. Each merge point then maps to a CTE containing a conditional expression that selects the correct predecessor value.

In practice, this approach is counterproductive. The SSA-to-GSA conversion, while well-defined, requires a new intermediate representation for a single backend. More fundamentally, CTE subqueries in PostgreSQL defeat the properties that make the scalar expression approach viable: when materialized, the planner cannot push predicates into them or inline expressions across the boundary; when not materialized, the planner inlines them, which negates the sharing benefit entirely. Because effect hoisting (Chapter 5.3) already lifts row-independent computations out of filter bodies, the expressions that remain in PostgreSQL islands almost always reference per-row entity data; materialization therefore provides no sharing benefit, as each row triggers its own evaluation. The deduplication is also incomplete, as CTE subqueries share computation

only at join points while intermediate expressions within branches remain duplicated. Consequently, we do not pursue this approach [60, 65:18.3, 79].

A different approach compiles control flow into CTEs directly rather than deduplicating at join points. Tim Fischer, et al. [23] map each basic block to a CTE, with non-recursive chains for acyclic flow and a recursive CTE for loops, building on work by Denis Hirn and Torsten Grust [29] and Denis Hirn and Torsten Grust [30], who compile PL/SQL through SSA and ANF into recursive CTEs. Adapting this strategy for the MIR would require a dedicated pass to make inter-block dataflow explicit for a CTE-based backend – such as the aforementioned GSA translation, together with a separate code generation path. The CTE materialization and planner visibility limitations discussed above apply equally to this approach. The reported performance gains further rely on batched execution across multiple invocations via subquery decorrelation for correlated `LATERAL` joins, which PostgreSQL does not implement; without decorrelation, PostgreSQL executes each invocation independently. Because these limitations compound, we do not adopt this strategy; Chapter 10.1.2.4 discusses extending the PostgreSQL backend to broader control flow once they can be addressed.

Each compiled island returns a continuation: all the control-flow state needed to either filter the row or resume execution on another backend. When PostgreSQL can evaluate a filter to completion, no transfer to the runtime is necessary and the row can be accepted or rejected directly. When control flow exits the island, the continuation must carry the target block and the set of live-out locals so that execution can resume elsewhere. Both cases are encoded in a single composite row: `(filter boolean, block int, locals int[], values jsonb[])`. The `filter` field is three-valued: `TRUE` and `FALSE` indicate early termination, in which case all other fields are `NULL` and the row is eagerly filtered without leaving PostgreSQL. `NULL` indicates that a continuation on another backend is required, where `block` identifies which basic block to resume at and the `locals` and `values` arrays carry the live-out state. These two arrays are parallel rather than a fixed-width composite: the set of live-out locals is typically sparse relative to the full set of locals in a body, so parallel arrays avoid large spans of `NULL` values that a positional encoding would require. Algorithm 14 formalizes the compilation of each island into the corresponding continuation expression.

Values that cross the interpreter-PostgreSQL boundary require an encoding into a type that PostgreSQL understands. As established in Chapter 7.2.1, the format chosen is JSON. Because HashQL contains types that cannot be represented in JSON, a semantic collapse is necessary, which introduces ambiguity. Figure 14 details this collapse: multiple HashQL types map to the same JSON representation.

$$
\begin{aligned}
S &: \text{HashQL Type} \to \text{JSON Type} \\
S(\text{Boolean}) &= \texttt{boolean} \\
S(\text{Integer}) &= \texttt{number} \\
S(\text{Float}) &= \texttt{number} \\
S(\text{String}) &= \texttt{string} \\
S(()) &= \texttt{null} \\
S((T_1, ..., T_n)) &= [\ S(T_1)\ , ..., S(T_n)\ ] \\
S(\text{List}\langle T\rangle) &= [\ S(T)\ , ...\ ] \\
S(\langle k_1 : T_1, ..., k_n : T_n\rangle) &= \{\ k_1 : S(T_1)\ , ..., k_n : S(T_n)\ \} \\
S(\text{Dict}\langle K, V\rangle)^{\text{f}} &= \{\ K : S(V)\ , ...\ \} \\
S(\text{Opaque}(T)) &= S(T) \\
S(\text{Closure}) &= \bot
\end{aligned}
\tag{23}
$$

Figure 14: Type-directed JSON serialization.

Serialization is therefore lossy: the deserializer must use out-of-band type information to reconstruct the original value. The MIR retains full type information at every consumption site, which guides deserialization: each value is accompanied by its expected type, and upon recursion into nested structures the type is followed as well. Union types present the only complication, as multiple deserialization paths are possible. The deserializer checks every variant until one matches; Chapter 7.2.1 guarantees that each union is unambiguous, which ensures that no value is classified as the wrong type.

Temporal metadata requires separate treatment: PostgreSQL stores temporal ranges as `tstzrange`, which has no JSON equivalent. All temporal intervals in the system follow a canonical left-closed right-open form: the start bound is always inclusive, while the end bound is either exclusive or unbounded. The compiler decomposes each range into a struct with `start` and `end` fields, where `start` is an epoch-millisecond integer and `end` is either an epoch-millisecond integer or `null` for unbounded ranges. This preserves overlap-check semantics inside PostgreSQL while allowing lossless reconstruction by the runtime.

Compilation of individual statements into SQL expressions is largely mechanical: each MIR rvalue maps to its SQL counterpart, and operands are resolved through the expression map L. The one complication is that values extracted from JSONB are untyped from PostgreSQL's perspective. Arithmetic operations therefore require explicit casts to the expected numeric type, and equality comparisons wrap both operands in `to_jsonb` to ensure type-aware comparison rather than textual matching. Vertex path accesses resolve to entity table columns, environment captures to prepared statement parameters, and remaining projections to JSONB paths extracted through `json_extract_path`. The control-flow compilation in Algorithm 14 invokes this expression translation at each statement before dispatching on the block's terminator.

Algorithm 14: Island compilation

[f] Non-string keys are rejected at placement (Chapter 7.2.1).

```
function CompileIsland(island)
    L ← ∅
    return CompileBlock(entry(island), island, L)
end

function CompileBlock(b, island, L)
    for l = r in statements of b do
        L[l] ← compile r using L
    end
    match term(b)
        case Goto(b′, a) where b′ ∈ island then
            bind a to parameters of b′ in L
            return CompileBlock(b′, island, L)
        case Goto(b′, a) where b′ ∉ island then
            return IslandExit(b′, live-out(b, L))
        case Return(v) then
            return ReturnFilter(coalesce(L[v], false))
        case SwitchInt(d, T) then
            ▷ T includes all explicit targets and the otherwise target
            e ← compile d using L
            arms ← ⟨null → ReturnFilter(false)⟩
            for (v_i, b_i) in T do
                if b_i ∉ island then
                    append v_i → IslandExit(b_i, live-out(b, L)) to arms
                    continue
                end
                L′ ← snapshot L; bind target args to b_i params
                append v_i → CompileBlock(b_i, island, L′) to arms
            end
            return Case(e, arms)
    end
end
```

PostgreSQL's `NULL` semantics require defensive handling at decision points. In a well-typed program with conformant data, a `NULL` value cannot arise during pipeline operator evaluation: the type system guarantees that accessed properties exist. The storage layer is not under the type system's control, however, and a missing JSONB key produces `NULL` rather than an error. Because `NULL` is infectious in SQL, it would propagate silently through arithmetic and comparisons and produce an incorrect filter decision. The compiler therefore guards at each decision point (Algorithm 14): every control-flow expression includes a `NULL` check as its first condition that rejects the row if the discriminant is `NULL`, and every `Return` continuation wraps the filter value to coalesce `NULL` to `FALSE`. Intermediate expressions are left unguarded, as `NULL` propagation through them is correct: SQL `NULL` appears only in the continuation's placeholder fields, while the HashQL unit value, encoded as JSON `null`, is transferred as a valid value.

The compiled continuation is a row value, but its fields serve different consumers: `filter` is used in the WHERE clause for early termination, while the remaining fields are returned in the SELECT list for the runtime to decode. A subquery in the SELECT list would suffice, but PostgreSQL expands composite field access at parse time, repeating the full expression once per field rather than evaluating it once. The planner does not perform common subexpression elimination, so the cost multiplies with the number

of accessed fields. Each island is therefore placed in a `CROSS JOIN LATERAL` subquery. `LATERAL` is necessary because the continuation expression references columns from the base table. `CROSS JOIN` is correct because every candidate row must be evaluated; there is no unmatched case.

Even through a `LATERAL` subquery, the planner may still inline the expression and reintroduce the duplication. PostgreSQL's `OFFSET 0` acts as a materialization fence that prevents this inlining, and the planner's cost estimate favors the fenced plan (Appendix A). Empirical measurement qualifies this expectation: for queries without conditional branches, the fence produces a modest improvement, but for queries with conditional CASE trees the materialization overhead exceeds the duplication cost by a wide margin, with fenced execution $3\times$ to $4\times$ slower than unfenced. The generated queries therefore omit the fence and accept the planner's inlining decisions [65:8.16.5].

```sql
SELECT
    -- continuation fields per island
    (i1.c).block,  (i1.c).locals,  (i1.c).values,
    (i2.c).block,  (i2.c).locals,  (i2.c).values,
    -- entity path columns from provides set
    <path expressions>
FROM <head table> AS t
    -- one LATERAL per PostgreSQL island
    CROSS JOIN LATERAL (
        SELECT <CASE tree>::continuation AS c
    ) AS i1
    CROSS JOIN LATERAL (
        SELECT <CASE tree>::continuation AS c
    ) AS i2
    -- entity path joins added lazily
    <LEFT JOIN ...>
WHERE
    -- temporal overlap conditions
    t.transaction_time && $txn
    AND t.decision_time && $dec
    -- per-island filter
    AND (i1.c).filter IS NOT FALSE
    AND (i2.c).filter IS NOT FALSE
```

Listing 20: Generated query template for two compiled islands.

The full query assembles these components around the base table (Listing 20). The graph read head determines the starting table and constrains it through temporal overlap conditions on the transaction-time and decision-time axes. Each pipeline body contributes its islands as `LATERAL` subqueries to the FROM clause and a filter condition to the WHERE clause. The SELECT list returns the continuation fields for the runtime to decode. Data required across islands (Chapter 7.7) is requested through additional SELECT columns, with table joins added lazily as the referenced entity paths demand.

8.3. Orchestrator

The orchestrator currently supports one suspension type: graph reads. Each suspension type carries its own fulfillment and continuation strategy; adding a new effectful terminator requires a suspension variant and a corresponding fulfillment strategy, and no structural changes to the orchestrator.

For a graph read, fulfillment begins at the authoritative source of temporal-to-vertex validity data: currently PostgreSQL. The head operation determines the candidate set by constraining the temporal slice and vertex type. Because PostgreSQL is both the authoritative store and a valid execution backend, the head query and all PostgreSQL island computation are fused into a single prepared statement (Chapter 8.2), parameterized from the interpreter's current state and executed as one round-trip. The fusion is sound because the transition model enforces an invariant: PostgreSQL islands may only be exited, never entered (Chapter 7.3). Because re-entry is forbidden, all reachable PostgreSQL islands form an entry-side prefix of any execution path through a pipeline body, so their computation can be embedded in the initial query.

Each row returned by the query passes through the pipeline's chain of operations, currently only filter operations. The orchestrator proceeds block by block: given a block, it identifies the current island, which determines the execution mechanism. For a PostgreSQL island, the orchestrator checks the continuation state decoded from the result row. If the continuation's block field is `NULL`, the row has already passed the corresponding WHERE condition: early termination returned `TRUE`, so the filter is implicitly satisfied. If a continuation exists, the orchestrator writes the decoded target block and live locals into the interpreter's callstack and advances execution to the continuation point.

For an interpreter island, the orchestrator runs the interpreter until either a value is returned or a transition to another island occurs. If a nested graph effect is encountered during execution, the orchestrator fulfills it recursively through the same suspension protocol, so the dispatch requires no special case for nesting depth.

The result of any filter expression inside the pipeline is a HashQL integer with a width of one, interpreted as a boolean. Only when all filters in the pipeline accept is a vertex admitted. The tail strategy then determines how to accumulate accepted vertices; the current implementation collects them into a list. The collected output becomes the continuation value for the original suspension, and the interpreter resumes at the graph read's successor block.

Vertex hydration is demand-driven: instead of loading the complete vertex in a single operation, the orchestrator populates only the fields that downstream islands have requested. Each island's requires set (Chapter 7.7) is resolved against provider islands during placement, and the resulting provides sets determine which vertex paths are made available at each island boundary. Partial hydration is therefore transparent to consumers: the type system prohibits access to fields outside the known set, traversal-aware path analysis (Chapter 6.2) determines which of those fields are actually reached, and the island dependency contract guarantees that every such field has been populated by a prior provider.

9. Evaluation

The compilation pipeline presented in Chapter 5 through Chapter 8 produces heterogeneous, multi-backend execution from a single functional source. Three properties remain undemonstrated: that dispatch across backends produces correct results, that the placement solver's decisions improve execution over naive single-backend strategies, and that the compilation overhead is justified by the execution gains.

9.1. Methodology

Benchmarking of Rust code uses the Criterion framework, which provides bootstrap confidence intervals on the estimated mean rather than point estimates. Each benchmark first runs a warmup phase to stabilize CPU and cache state before recording statistically significant data. The measurement phase then collects N samples, where each sample executes $i * d$ iterations and records the total elapsed time; per-iteration time is the quotient of elapsed time and iteration count. The scaling factor d is determined during warmup. Outliers are flagged through a modified Tukey method but retained in the dataset [1].

Benchmarking of equivalent Python code for the interpreter comparison uses the built-in `timeit` module with manual statistical collection, run on CPython 3.8. Newer CPython versions introduce a JIT that would distort the comparison against a non-JIT interpreter.

The PostgreSQL backend operates against a minimal subgraph containing five entities across two entity types, including draft and link entities that exercise temporal overlap predicates. The PostgreSQL server runs inside Docker with fsync and log statements disabled to minimize noise and maximize throughput; the container is reused across samples to avoid startup overhead.

All benchmarks run on an Apple M5 Max with 128 GiB, macOS Tahoe 26.4.1, Rust 1.97.0-nightly (2026-04-09), and PostgreSQL 17.

9.1.1. Corpus

The benchmark corpus consists of five query shapes chosen to span the MIR's control-flow and placement diversity. Where feasible, each query is re-implemented as a hand-crafted Filter API equivalent, the primary interface to the graph prior to this work. The full J-Expr source of each query is provided in Appendix C.

1. **All entities.** A single always-true filter. Produces one basic block assigned entirely to PostgreSQL. The simplest case: no branching, no input parameters, no backend transitions.
2. **Filter by UUID.** Single equality comparison against an input parameter. Still a single block on PostgreSQL, but exercises input binding and parameter passing through the prepared statement.
3. **Diamond CFG.** Conditional filter with if-then-else, producing a diamond CFG. The solver assigns the comparison block to the interpreter because it operates on `EntityId`, a composite type containing `Option<T>` that the PostgreSQL backend cannot serialize (Chapter 7.2.1). This is the simplest heterogeneous case.
4. **Sequential filters.** Two chained filters, producing multiple bodies. All blocks are assigned to PostgreSQL. Exercises pipeline composition across graph effect boundaries.
5. **Stress test.** Three sequential filters combining boolean connectives and conditional trees. The most complex query in the corpus, with multiple bodies and heterogeneous placement driven by the same `EntityId` constraint as the diamond CFG.

9.2. Correctness

Correctness is verified through a test suite spanning every compiler crate. Branch coverage exceeds 80% as measured by llvm-cov [22].

Crate	Unit	Snapshot	Compiletest	Doc
`hashql_ast`	5	0	232	0
`hashql_compiletest`	46	0	0	0
`hashql_core`	862	7	0	389
`hashql_diagnostics`	20	0	0	120
`hashql_eval`	53	25	75[g]	0
`hashql_mir`	461	218	108	42
`hashql_hir`	0	0	167	0
`hashql_syntax_jexpr`	83	406	0	0
Total	**1530**	**656**	**582**	**551**

Table 6: Test counts per compiler crate.

The tests fall into four categories. Unit tests are isolated per module and validate specific invariants through assertions, without requiring the rest of the pipeline. An example is verifying that Tarjan's algorithm correctly identifies strongly connected components in a constructed graph.

Compiletests exercise the pipeline end to end. Each test is defined as a J-Expr query that is successively lowered through the relevant compiler stages. Correctness is verified through snapshot integrity: the test harness records the output of each stage into a committed snapshot file, and any deviation from the expected output causes a failure in

[g] Includes 14 J-Expr and 2 programmatic tests executed against a live PostgreSQL instance.

CI. Because the snapshots are checked in, regressions surface as diffs in version control before they reach a reviewer.

Snapshot tests use the native representation of their crate directly, rather than lowering from J-Expr. The trade-off is higher control at the cost of representativeness: because the MIR after optimization is far removed from the original source, constructing a representative body from J-Expr alone is difficult, particularly for constructs the MIR supports but the HIR currently forbids, such as loops and additional binary operators. Snapshot tests consequently cover transformations that compiletests cannot reach, particularly MIR constructs the HIR does not yet emit.

Compiletests and snapshot tests cover every stage that does not require a database. For stages that do, a separate harness runs the complete pipeline through to the orchestrator, dispatching against a live PostgreSQL instance. These tests use a different harness because they require side effects, database setup and teardown, that the standard compiletest framework intentionally does not support. Inputs are J-Expr queries where possible, with programmatically constructed bodies as a fallback. These tests verify semantic correctness of heterogeneous execution by comparing the orchestrator's output against expected results for each query.

9.3. Compilation and Execution Performance

The compilation pipeline spans nine instrumented phases, from parsing through code generation. Figure 15 decomposes the total compilation time for the simplest query in the corpus. To understand where time is spent within the HIR lowering stage developed as part of the Großer Beleg [11], the figure further breaks that stage into its constituent transformations, type inference, and type checking. The four phases this thesis contributes, MIR reification (Chapter 5.5), MIR transformation (Chapter 6.3), execution analysis (Chapter 7), and code generation (Chapter 8), together account for 0.72% of the total. The HIR/ANF normalization work described in Chapter 5 is subsumed within the HIR phases, as reification from the normalized HIR/ANF into the MIR is where the thesis-specific timer begins [11].

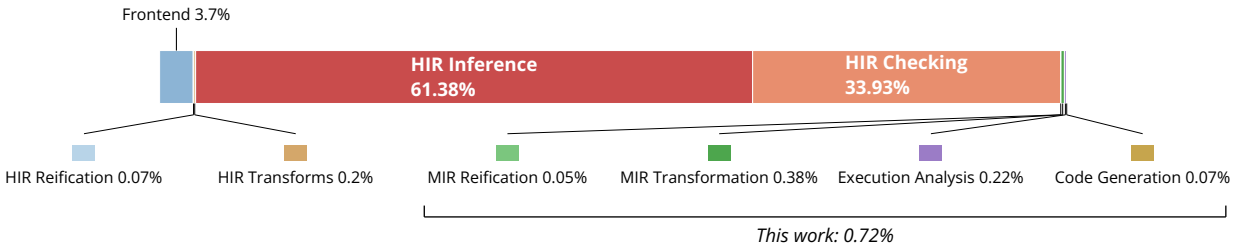


Figure 15: Compilation time breakdown for the all-entities query.

HIR type inference and HIR type checking together account for 95.3% of the pipeline. HashQL's type system is structural with nominal opt-in, so type checking is not constrained to identity verification but makes extensive use of subtype checking. Mature type systems amortize repeated subtype verification through caching; the HashQL type system was designed with such a mechanism in mind, but its implementation has not been completed. Because we focused on the MIR and heterogeneous execution rather than type system performance, the solver and checker remain unoptimized. The cost

also reflects the breadth of inference: large parts of the query leave types annotated as _, and the type system resolves them through structural subtyping.

Within the thesis contribution, the cost is proportional to the work each phase performs. MIR reification and code generation are the cheapest phases: reification translates mechanically from the ANF to SSA equivalence (Chapter 5), while code generation compiles each island in a single traversal through the case-based scalar expression strategy (Chapter 8.2), avoiding the overhead of CTE-based approaches. MIR transformation accounts for the largest share, reflecting the rich set of optimization passes (Chapter 6.3) applied to reduce the search space for subsequent analysis. Execution analysis, despite solving a constraint problem over the CFG (Chapter 7.5), consequently takes less time than the transform passes that simplify its input.

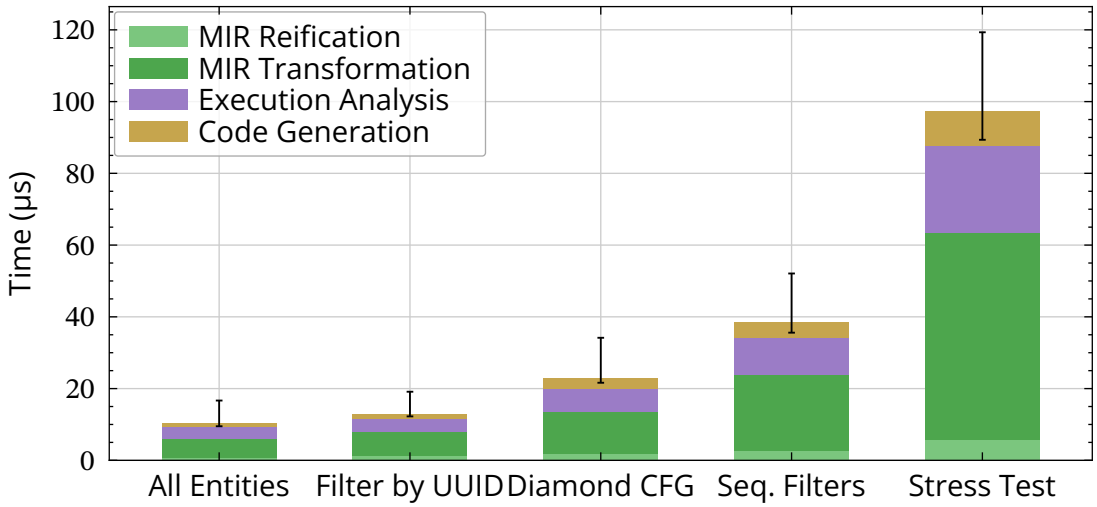


Figure 16: Thesis contribution to compilation time across query shapes.

Figure 16 confirms this across all five queries: the time each phase consumes scales roughly linearly with query complexity, and the total thesis contribution remains well beneath $100\mu s$ even for the most complex query in the corpus.

The compilation pipeline produces artifacts akin to binaries in traditional compilers: a PostgreSQL prepared statement for each PostgreSQL island, the MIR body for the interpreter, and the island dependency graph that coordinates between them. These artifacts are sufficient to execute the query at runtime; once code generation has completed, compilation does not need to be repeated.

HashQL differs from languages like Rust in that it must accommodate both interactive, script-like use and compiled scenarios that were not possible before this work. The compilation pipeline is therefore designed with both modes in mind. If compilation latency becomes perceptible, the scripting model breaks down; the measurements above confirm that the thesis phases do not introduce that risk. Compilation also pays for itself immediately: because the compiled pipeline's execution time is substantially lower than the Filter API's (Chapter 9.4), the break-even point is below one invocation for all-entities and filter-by-uuid, and approximately one invocation for the stress test.

9.4. Filter API Comparison

The Filter API is the runtime interface that preceded HashQL and served as the compilation target during the Großer Beleg. It exposes a condition AST that translates directly to SQL. The Filter API supports only a subset of the operations HashQL can express: both can represent a single graph read filter effect, but where HashQL composes multiple effects, the Filter API is limited to a single invocation. To compare the two, we converted the subset of the corpus that is expressible in both into the Filter API [11].

Not all queries in the corpus are portable. Composite field access is not available in the Filter API, control flow can only be expressed through boolean decomposition (`all` / `any`), and only a single filter is supported, with multiple filters combined through conjunction. These constraints exclude the diamond CFG and sequential queries from the comparison.

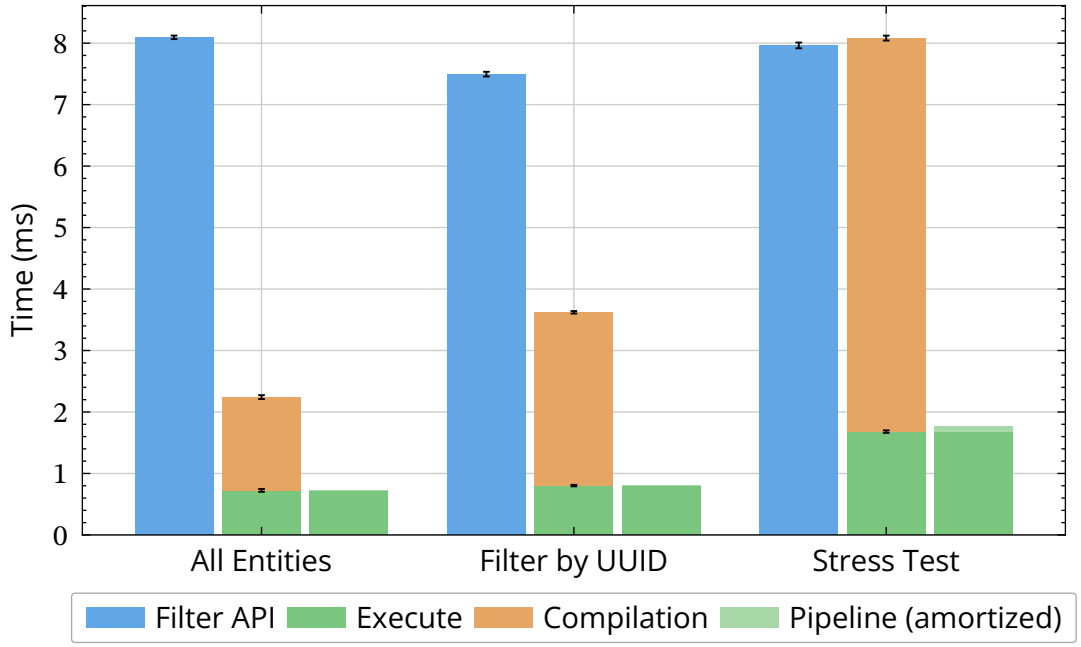


Figure 17: Filter API versus compiled pipeline execution time, with and without type system overhead.

Figure 17 decomposes the cost into three components: full compilation as a one-time cost, the MIR pipeline in isolation, and execution time. The Filter API is near-constant across query shapes because it dispatches directly to PostgreSQL without a compilation step. Compilation time in the pipeline scales with query complexity, consistent with the attribution in Chapter 9.3, but the MIR contribution remains a small fraction of the total. The stress test shows increased execution time attributable to interpreter overhead, as discussed in Chapter 9.6.

The comparison is asymmetric in two respects. The Filter API performs authorization checking, which this work does not implement but supports structurally through additional filter bodies and property object masking. Conversely, the compiled pipeline returns substantially less data through partial hydration (Chapter 8.3), reducing both transfer volume and deserialization cost. The same mechanism would extend to selective returns and mapping functions once implemented (Chapter 10.1.2.1).

Both approaches produce the same SQL structure: table joins where required and a `WHERE` clause for filtering. The compiled pipeline, however, applies the full MIR transformation and execution analysis pipeline to simplify the query before emission, whereas the Filter API translates its condition AST without optimization.

9.5. Placement Effectiveness

The placement solver (Chapter 7.5) assigns each basic block to an execution backend based on capability constraints and the cost model. The baseline for comparison forces all blocks onto the interpreter, bypassing PostgreSQL entirely, by setting all statement costs to an infeasible value ($9,000,000$); the difference isolates the effect of the solver's placement decisions. The benchmark uses an ambient dataset of 10240 entities to amplify the difference between database-side and interpreter-side filtering.

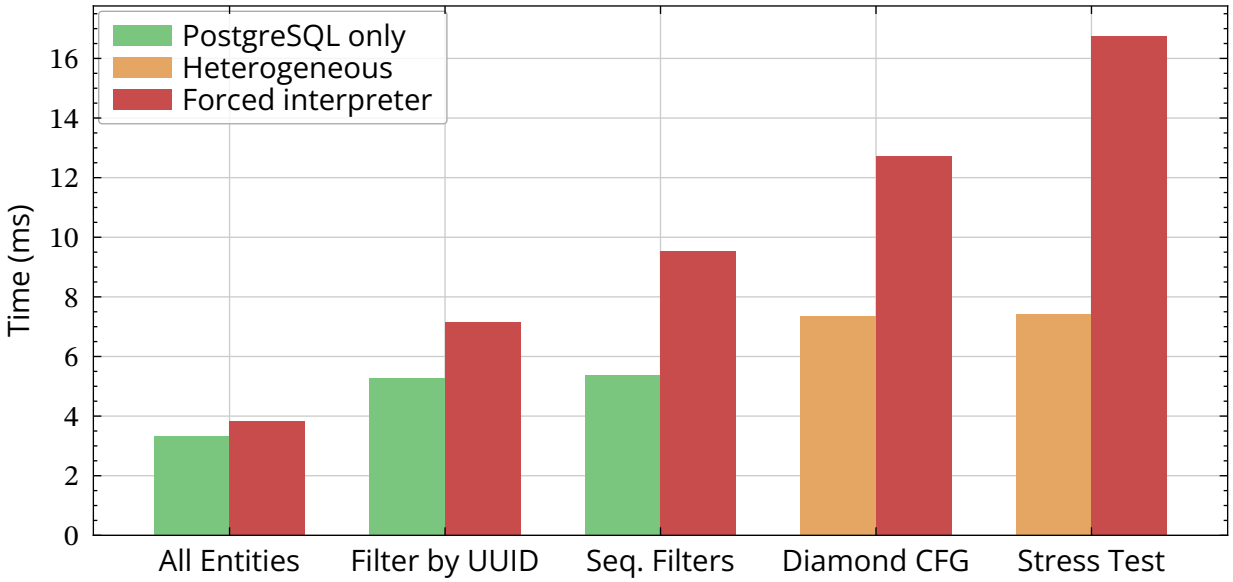


Figure 18: Execution time with solver placement versus forced interpreter execution.

Figure 18 shows the result across all five queries. The regression from solver placement to forced interpreter ranges from $+16\%$ for all-entities to $+126\%$ for the stress test. The gap scales with query complexity: queries that push more filtering into PostgreSQL benefit more from the solver's placement, because PostgreSQL evaluates predicates inside the database engine rather than transferring rows to the interpreter for evaluation.

The solver produces non-uniform placement in two of the five queries without any manual annotation. In the diamond CFG, the solver assigns two blocks to PostgreSQL but routes a single block to the interpreter because it compares against an `EntityId`, a composite type containing `Option<T>`, which the PostgreSQL backend cannot serialize (Chapter 7.2.1). In the stress test, the same constraint produces two interpreter blocks within a predominantly PostgreSQL body. The solver handles the resulting backend transitions through the island dependency graph (Chapter 7.7), with transfer costs accounted for by the cost model.

The non-uniform queries do not show a disproportionate execution time relative to the pure PostgreSQL ones, despite requiring interpreter involvement. Because the contin-

uation model (Chapter 8.2) filters rows within the PostgreSQL islands before control reaches the interpreter, the interpreter processes only rows that survive the preceding predicates. The cost of the backend transition is consequently bounded by the selectivity of the preceding PostgreSQL islands rather than by the total row count.

For the remaining three queries, all blocks are assigned to PostgreSQL. The forced-interpreter baseline still regresses by $+16\%$ to $+77\%$, because PostgreSQL evaluates the compiled `WHERE` clause across all rows in a single query execution, while the interpreter must tree-walk every row individually.

9.6. Interpreter Performance

To guarantee a feasible assignment for every basic block, we employ a tree-walking interpreter over the MIR as the universal fallback (Chapter 8.1). While the interpreter supports coroutine suspension to interact with the runtime, the following benchmarks elide this property and measure pure execution over four algorithms:

- **Recursive Fibonacci.** Exponential call tree; exercises frame allocation and arithmetic.
- **Ackermann.** Deep recursion with minimal per-call computation.
- **Bubble Sort.** Tight loop over array indexing, comparison, and swap. Always worst-case (reverse-sorted input).
- **Recursive Sum.** Linear recursion with one addition per frame.

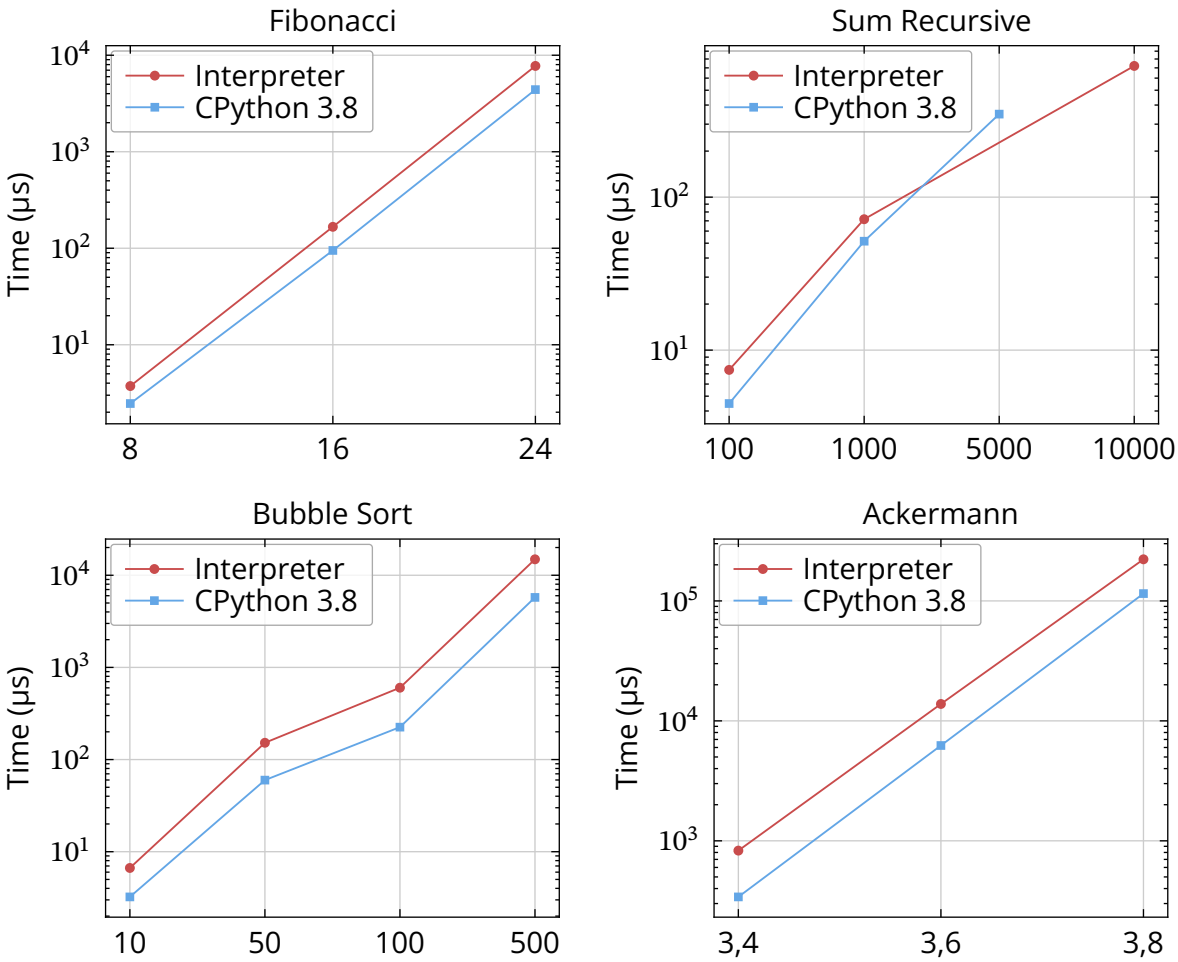


Figure 19: Interpreter versus CPython 3.8 across four algorithms at increasing input sizes.

Figure 19 compares the results against CPython 3.8, a well-established interpreter baseline. We chose this version specifically because it predates CPython's JIT introduction, which would confound the comparison. The interpreter is consistently $1.5\times$ to $2.7\times$ slower, with the ratio stable within each algorithm. The parallel lines on the logarithmic scale confirm that both follow the same asymptotic complexity; the difference is a constant multiplicative factor in per-operation cost.

The constant factor follows from the dispatch model. CPython compiles to bytecode and executes through a flat dispatch loop. The HashQL interpreter instead pattern-matches over the MIR instruction structure at each step, resolves operands through the local store, and dispatches terminators through the CFG. This indirection is inherent to tree-walking interpreters and is the primary target for a future register-based bytecode VM (Chapter 10.1.1.2).

Bubble sort shows the highest ratio ($\approx 2.6\times$) because the interpreter operates on persistent lists, which incur higher per-access cost than the contiguous C arrays CPython uses internally. Fibonacci and recursive summation show lower ratios ($\approx 1.5\times$ to $1.9\times$) because function call overhead dominates, where frame allocation cost is comparable between the two.

Recursive summation is the only algorithm where CPython cannot complete all input sizes: at $n = 10000$, CPython exhausts the native call stack and terminates with a segmentation fault. The interpreter continues because its stack frames are heap-allocated, independent of the native call stack depth.

A register-based bytecode VM (Chapter 10.1.1.2) would reduce the constant factor, but the current $\approx 2\times$ gap against a pre-JIT CPython already confirms the interpreter as an adequate universal fallback.

10. Conclusion and Future Work

This work extends the HashQL compiler from a system that could parse and type-check queries into one that executes them across heterogeneous backends. The contribution comprises four stages: ANF normalization bridges the tree-structured HIR to a statement-based representation (Chapter 5); a type-preserving SSA-form MIR replaces the functional representation with a CFG of basic blocks suitable for transformation and analysis (Chapter 6); a solver over a separable cost model assigns each basic block to a backend under capability and transfer-cost constraints (Chapter 7); and a runtime phase first compiles each assigned island into backend-specific artifacts, then dispatches them through an orchestrator and a tree-walking interpreter that supports coroutine-like suspension for effectful computations (Chapter 8). Vertex hydration is demand-driven: only the fields that downstream islands require are fetched.

Two design decisions made this pipeline tractable. First, types are preserved throughout the MIR and transformed alongside the program: they guide optimization passes, drive size estimation for transfer cost, and determine capability constraints through serialization ambiguity across the differing computational models of the backends. Second, graph effects are modeled as first-class terminators whose suspension semantics allow the same representation to serve transformation, execution analysis, and runtime dispatch uniformly. Because effects occupy the same structural position as branches and returns, most standard compiler passes, dataflow analysis, liveness, dead code elimination, and inlining, operate on graph-effectful bodies without special-casing. The exceptions are passes that inspect target edges directly, such as SSA repair and CFG simplification, which treat effectful terminators as opaque boundaries.

The MIR uses a traditional compiler instruction set rather than relational algebra. This choice has two consequences: standard optimization techniques apply directly, and the traversal-based nature of the language, as motivated in the Großer Beleg [11], maps naturally to an imperative instruction model rather than a declarative one. HashQL is consequently represented as an ordinary typed programming language with first-class graph effect support. The per-instruction granularity is what makes statement-level placement analysis tractable: the same dataflow framework drives both traditional optimizations and execution-specific analyses such as size estimation and supported-statement propagation.

The cost model is the central integration challenge of this work: it must combine two distinct cost components, computational cost and transfer cost, into a single metric that

the solver can optimize. The ideal formulation accounts for shared fetch across same-target islands, but this introduces higher-order submodular cliques that combine with asymmetric non-metric pairwise costs and cycles in the CFG to produce an intractable optimization problem. A separable approximation reduces the problem to classical pairwise energy minimization by charging each block independently (Chapter 7.1). The approximation over-counts along same-target chains, but does so uniformly: no structural region of the CFG receives preferential treatment, which preserves the solver's ability to compare assignments without bias. The over-approximation is not neutral: it biases the solver toward colocating operations with the data they consume, because non-origin assignments are uniformly penalized.

The evaluation confirms three properties of the pipeline. The compilation overhead of the four thesis phases is negligible: MIR reification, MIR transformation, execution analysis, and code generation together account for 0.72% of the total pipeline, and the compiled artifacts amortize within a single invocation. Because compilation cost is negligible relative to execution time, the pipeline remains viable for interactive, script-like use. The same property admits adaptive cost tuning (Chapter 10.1.1.1). The placement solver produces measurably better assignments than a single-backend strategy: speedups range from $1.2\times$ to $2.3\times$ over forced-interpreter execution across the corpus. The solver discovers heterogeneous placement from capability constraints alone, without manual annotation: the cost model and eligibility analysis are sufficient to route operations to the appropriate backend. The interpreter serves as an adequate universal fallback at a constant factor of approximately $2\times$ relative to CPython 3.8. This confirms the viability of a tree-walking design as a baseline; the constant-factor gap motivates the bytecode VM discussed below.

The central limitations fall into two categories. The current implementation covers a single pipeline operator (filter), executes read-only queries, and provides two runtime backends; the embedding target participates in analysis but has no execution engine. These are scope boundaries that the architecture is designed to absorb. Adding a pipeline operator, for example, requires a body provenance, a terminator variant, and backend-specific compilation rules; the solver, orchestrator, and island model remain unchanged. The deeper limitations concern placement quality. The cost model operates in open loop: costs are statically determined with no feedback from observed execution, so inaccuracies in the cost entries propagate into suboptimal assignments. No branch frequency or loop iteration data is available; reasoning over looping regions is consequently a best-case approximation that cannot account for the multiplicative effect of per-iteration transitions. The separable approximation, by contrast, is a deliberate design choice whose trade-offs are argued in Chapter 7.1 and are not among the limitations identified here. The open-loop and missing-frequency constraints are architectural rather than fundamental: the system is designed to accept feedback through configurable cost entries, and closing that loop would address both directly.

10.1. Future Work

10.1.1. Execution Infrastructure

The placement solver and interpreter are the two runtime components most directly constrained by the limitations identified above. Closing the cost model's feedback loop and replacing the tree-walking interpreter address the two most immediate structural bottlenecks; improving the solver's reduction capabilities is a longer-term refinement.

10.1.1.1. Adaptive Cost Tuning

The placement solver operates on static cost entries that are configured once and never revised. If these entries diverge from observed execution, the solver's assignments degrade without any corrective mechanism. The architecture already provides the prerequisites for closing this loop. Costs are configurable parameters, not hardcoded constants, and the solver is a pure function from costs to assignments. Re-running execution analysis through code generation accounts for 0.29% of the total pipeline, so the overhead of re-optimization is negligible.

An adaptive system would execute a query under the current assignment, measure per-island execution times and transition overheads, and adjust the cost entries toward observed values. The solver then re-computes the assignment against the updated model and recompiles only the affected islands. Iteration continues until the assignment stabilizes or the cost delta falls below a threshold. Because the solver is deterministic given fixed costs, stability of the cost entries implies stability of the assignment; whether a particular update rule converges is an open question that depends on the measurement noise and the feedback dynamics.

The loop iteration problem identified in the conclusion would resolve naturally under this scheme, as the measured cost of a loop-interior transition already reflects the iteration count.

This is an extension of the current infrastructure, not a replacement: the same solver and static cost model produce the initial assignment, but frequently executed queries could be iteratively refined over successive invocations.

10.1.1.2. Register-based Bytecode VM

The tree-walking interpreter incurs a constant overhead of approximately $2\times$ relative to CPython 3.8, attributable to per-instruction pattern matching over the MIR instruction structure, operand resolution through the local store, and terminator dispatch through the CFG. A register-based bytecode VM would replace this with a flat dispatch loop over a compact instruction encoding, where operands reference virtual registers rather than named locals. A register-based representation also opens optimization opportunities that the current value-based SSA form forecloses. The MIR uses value-based SSA because the upstream language is immutable, but this forces successive modifications to a struct to reconstruct the entire value each time. Under a register-based representation, these could be merged into in-place updates. Beyond single-threaded performance, the current interpreter is not thread-safe, so pipeline bodies process rows sequentially. A bytecode VM with an appropriate value representation would

enable parallel evaluation of rows within a single pipeline body, a natural fit given that each row is independent.

10.1.1.3. PBQP Solver

The separable cost model from Equation 11 maps directly to PBQP: both involve a graph with a small label set per node and pairwise costs on edges, with the objective of minimizing total cost. PBQP admits R1 and R2 reductions that fold degree-1 and degree-2 nodes into the cost vectors of their neighbors without information loss; empirically, these reductions eliminate most nodes in register allocation problems before any heuristic runs. Applied to the placement problem, reductions would shrink the CFG the solver receives; the smaller graph reduces solver runtime and improves assignment quality [28, 74].

10.1.2. Pipeline and Backend Extensions

The current pipeline supports a single graph operation (filter) across two execution backends.

10.1.2.1. Pipeline Operators

The current implementation supports one head, one body, and one tail operator respectively (Chapter 6.1). The pipeline accommodates additional operators without structural changes to the compilation or placement infrastructure. Future extensions include: querying for the full set of vertex kinds, including property types and data types rather than entities alone; mapping data from sources onto a new shape, which enables selective returns; creating sorted views; and concluding an effect by counting rather than collecting. The set of operations is analogous to existing combinator libraries such as Rust iterators.

Each new body operator requires a body provenance, a terminator variant, backend-specific compilation rules, and an entry in the statement placement cost table; the rest of the infrastructure requires no modification.

10.1.2.2. Nested Graph Effect Queries

Effectful terminators run exclusively on the interpreter, because they require runtime coordination that no compiled backend can provide. An alternative is to dispatch effectful terminators to other backends when the inner pipeline is fully expressible on that target. A graph traversal inside a PostgreSQL body could, for example, be fused into the outer query as a `JOIN` instead of returning to the orchestrator. The precise conditions under which this fusion is sound remain to be determined.

10.1.2.3. Write Operations

The current execution model is read-only: graph effects query the bi-temporal graph but do not modify it. Write operations introduce two requirements absent from reads. First, catastrophic failure during a write sequence must roll back all preceding writes within the same transaction; the read path requires no such mechanism because referential transparency makes re-execution safe [11]. Second, thunking and inlining may duplicate effectful expressions that must execute exactly once; the orchestrator would need to memoize effect execution, for example through a propagated effect identifier, to prevent duplicate writes from reaching the store.

10.1.2.4. CTE-based PostgreSQL Compilation via GSA

The current PostgreSQL backend compiles control flow through scalar expression duplication, which cannot express loops (Chapter 8.2). Tim Fischer, et al. [23] demonstrate an alternative that allows the expression of arbitrary control-flow using CTEs. Chapter 8.2 argues against this approach for the current use case due to PostgreSQL's CTE materialization behavior and missing decorrelation support. Whether the approach can nonetheless outperform the interpreter for loop-heavy bodies, where the per-iteration cost of tree-walking is high, remains an open empirical question. GSA would provide the principled intermediate representation for such an implementation, replacing block parameters with γ-functions that carry the controlling predicate [60, 79].

10.1.2.5. Embedding Store

The embedding backend participates in placement analysis as a third target (Chapter 7.2) but has no runtime backend, because the HASH graph does not yet provide a dedicated vector store. Connecting an embedding store would validate the architecture at $k = 3$ backends end-to-end and enable vector similarity operations within the pipeline. The placement infrastructure requires no changes: cost entries and eligibility rules for the embedding target are already defined. The missing component is a code generation module that compiles embedding islands into vector store queries and an orchestrator extension that dispatches them.

10.1.3. Language and Type System

The HIR and the type system constrain which queries the compiler can accept and how their types interact during compilation.

10.1.3.1. HIR Pattern Matching

Conditional expressions in the HIR are limited to if-expressions, which lower to a two-armed `SwitchInt` terminator. Pattern matching would generalize this to multi-way branching with destructuring, lowering to a `SwitchInt` with one arm per pattern. Pattern matching is also a prerequisite for type refinement: narrowing a union type to a specific variant within a branch requires the pattern to carry the discriminating information. The MIR infrastructure already supports arbitrary `SwitchInt` fan-out; the missing component is a HIR-level pattern language and the desugaring pass that compiles match expressions into the existing terminator.

10.1.3.2. Unification of Empty Value Representations

The type system introduced in the Großer Beleg represents empty values through multiple overlapping constructs: a `Null` primitive type, the empty tuple, and empty closed structs. This redundancy complicates every consumer of the type system, from inference through serialization. A unified representation would remove `Null` as a primitive type in favor of the empty tuple and disallow empty closed structs. Whether empty open structs should remain representable or be subsumed into closed structs through the subtyping relationship is an open design question [11].

10.1.4. Optimization

The canonicalization loop (Chapter 6.3.2) applies a fixed set of passes before and after inlining.

10.1.4.1. SwitchInt Flattening

Nested boolean conditions in filter predicates produce chains of two-armed `SwitchInt` terminators. When two switches are directly successive and neither carries an otherwise branch, CFG simplification could merge them by folding the discriminant into a composite index. Let v be the original discriminant, j the matched case of the first switch, $|N|$ the case count of the first switch, and r the target case in the second:

$$\delta = (v = j), \quad \mathrm{idx} = v + \delta * (|N| - v + r) \tag{24}$$

The folded index idx maps directly to a single merged `SwitchInt` that covers both case spaces. The trade-off is that each original switch requires only a comparison and a pointer update, while the merged version pays multiple arithmetic operations across all arms. Whether the overhead of the additional instructions outweighs the benefit of fewer blocks and transitions is an open question, particularly given that directly successive switches are infrequent in practice.

10.1.4.2. Positional Struct Field Access

Closed struct fields are accessed by name. Because closed structs have a fixed field order, the name can be resolved to a positional index during reification or a dedicated pass, which allows the MIR to treat closed structs and tuples uniformly. This eliminates the linear scan over field names that the interpreter currently performs on every struct access. The result is a constant-time positional lookup equivalent to tuple access. Open structs retain name-based access, as their field set is not fully known.

10.1.4.3. Sparse Conditional Constant Propagation

instruction simplification and copy propagation evolved as separate passes in the canonicalization loop, each performing a single forward traversal (Chapter 6.3.2). SCCP would unify both into a single worklist-driven fixpoint that simultaneously propagates constants and determines branch reachability. The combined pass discovers constants that neither pass finds independently: a branch whose condition is constant may render one arm unreachable, which eliminates definitions that the separate passes consider live. Integrating SCCP would reduce the number of canonicalization iterations required to reach a fixpoint [80].

10.1.4.4. Cross-Body Constant Propagation for Effectful Bodies

Constant propagation in the current pipeline is body-local: each body is analyzed independently, so a constant passed through a closure environment or a function argument is opaque to the callee. For effectful bodies, this limitation interacts with effect hoisting (Chapter 5.3): a constant hoisted out of a filter body must be transferred into the closure environment, incurring a transfer cost that in-body constant folding would have avoided. Cross-body propagation would specialize effectful bodies at their call sites by substituting known constants from the caller's context before execution analysis runs. This is predicated on each pipeline operator body being unique to its pipeline, which the current architecture does not guarantee; if bodies are shared across pipelines,

specialization would require duplication. The practical benefit is unclear; this direction is exploratory rather than a committed improvement.

10.1.4.5. Tail Call Optimization

Recursive functions allocate a new stack frame per call. When the recursive call is in tail position, it can be converted into a loop. Memory consumption drops from linear in recursion depth to constant. This transformation is most naturally expressed at the HIR level, where tail position is syntactically apparent, rather than recovering it from the MIR after lowering.

Bibliography

[1] 2026. criterion-rs/criterion.rs Retrieved from https://github.com/criterion-rs/criterion.rs

[2] Christopher R. Aberger, et al. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Transactions on Database Systems* 42, 4 (December 2017), 1–44. https://doi.org/10.1145/3129246

[3] Alfred V. Aho, et al. (Eds.). 2007. *Compilers: principles, techniques, & tools* (2. ed., Pearson internat. ed.). Pearson Addison-Wesley.

[4] J. R. Allen, et al. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*, 1983. ACM Press, 177–189. https://doi.org/10.1145/567067.567085

[5] Andrew W. Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (April 1998), 17–20. https://doi.org/10.1145/278283.278285

[6] Chetan Arora and S.N. Maheshwari. 2014. Multi-label Generic Cuts: Optimal Inference in Multi-label Multi-clique MRF-MAP Problems. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014. IEEE, 1346–1353. https://doi.org/10.1109/CVPR.2014.175

[7] Chetan Arora, et al. 2012. Generic Cuts: An Efficient Algorithm for Optimal Inference in Higher Order MRF-MAP. In *Computer Vision – ECCV 2012*, Andrew Fitzgibbon et al. (eds.). Springer Berlin Heidelberg, 17–30. https://doi.org/10.1007/978-3-642-33715-4_2

[8] Omar Batarfi, et al. 2016. A distributed query execution engine of big attributed graphs. *SpringerPlus* 5, 1 (December 2016), 665. https://doi.org/10.1186/s40064-016-2251-0

[9] Alexander Baumstark, et al. 2023. Adaptive query compilation in graph databases. *Distributed and Parallel Databases* 41, 3 (September 2023), 359–386. https://doi.org/10.1007/s10619-023-07430-4

[10] Julian Besag. 1986. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 48, 3 (July 1986), 259–279. https://doi.org/10.1111/j.2517-6161.1986.tb01412.x

[11] Bilal Mahmoud. 2025. A Processing Pipeline for Querying Bitemporal Graph Databases. Großer Beleg.

[12] Benoit Boissinot, et al. 2011. A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs. In *Programming languages and systems*, Hongseok Yang (ed.). Springer Berlin Heidelberg, 137–154. Retrieved from http://link.springer.com/10.1007/978-3-642-25318-8_13

[13] Y. Boykov, et al. 2001. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23, 11 (November 2001), 1222–1239. https://doi.org/10.1109/34.969114

[14] Shaoyuan Chen, et al. 2025. Scaling Asynchronous Graph Query Processing via Partitioned Stateful Traversal Machines. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, May 19, 2025. IEEE, 918–931. https://doi.org/10.1109/ICDE65448.2025.00074

[15] Alvin Cheung, et al. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 16, 2013. ACM, 3–14. https://doi.org/10.1145/2491956.2462180

[16] Julia Chuzhoy and Joseph (Seffi) Naor. 2007. The Hardness of Metric Labeling. *SIAM Journal on Computing* 36, 5 (January 2007), 1376–1386. https://doi.org/10.1137/06065430X

[17] Keith D. Cooper and Linda Torczon. 2023. *Engineering a compiler* (Third edition ed.). Morgan Kaufmann Publishers, an imprint of Elsevier.

[18] Ron Cytron, et al. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991), 451–490. https://doi.org/10.1145/115372.115320

[19] S. M. Deen. 1985. The Architecture of a Generalised Distributed Database System - PRECI. *The Computer Journal* 28, 3 (March 1985), 282–290. https://doi.org/10.1093/comjnl/28.3.282

[20] David J. DeWitt, et al. 2013. Split query processing in polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, June 22, 2013. ACM, 1255–1266. https://doi.org/10.1145/2463676.2463709

[21] Jennie Duggan, et al. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (August 2015), 11–16. https://doi.org/10.1145/2814710.2814713

[22] Taiki Endo. 2026. taiki-e/cargo-llvm-cov Retrieved from https://github.com/taiki-e/cargo-llvm-cov

[23] Tim Fischer, et al. 2024. SQL Engines Excel at the Execution of Imperative Programs. *Proceedings of the VLDB Endowment* 17, 13 (September 2024), 4696–4708. https://doi.org/10.14778/3704965.3704976

[24] Cormac Flanagan, et al. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design*

and implementation, June 1993. ACM, 237–247. https://doi.org/10.1145/155090.155113

[25] Victor Giannakouris, et al. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)*, December 2016. IEEE, 452–461. https://doi.org/10.1109/BigData.2016.7840636

[26] Joe Groff and Chris Lattner. 2015. Swift Intermediate Language: A High Level IR to Complement LLVM Retrieved from https://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf

[27] Philipp Marian Grulich, et al. 2021. Babelfish: efficient execution of polyglot queries. *Proceedings of the VLDB Endowment* 15, 2 (October 2021), 196–210. https://doi.org/10.14778/3489496.3489501

[28] Lang Hames and Bernhard Scholz. 2006. Nearly Optimal Register Allocation with PBQP. In *Modular Programming Languages*, David E. Lightfoot and Clemens Szyperski (eds.). Springer Berlin Heidelberg, 346–361. https://doi.org/10.1007/11860990_21

[29] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, June 11, 2020. ACM, 2677–2680. https://doi.org/10.1145/3318464.3384678

[30] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *Proceedings of the 2021 International Conference on Management of Data*, June 09, 2021. ACM, 723–735. https://doi.org/10.1145/3448016.3457272

[31] Jiao Huang, et al. 2019. Cost-Minimizing Online Algorithms for Geo-Distributed Data Analytics. *IEEE Access* 7, (2019), 163515–163525. https://doi.org/10.1109/ACCESS.2019.2951682

[32] P. Z. Ingerman. 1961. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM* 4, 1 (January 1961), 55–58. https://doi.org/10.1145/366062.366084

[33] Hiroshi Ishikawa. 2009. Higher-order gradient descent by fusion-move graph cut. In *2009 IEEE 12th International Conference on Computer Vision*, September 2009. IEEE, 568–574. https://doi.org/10.1109/ICCV.2009.5459187

[34] Stefanie Jegelka and Jeff Bilmes. 2011. Submodularity beyond submodular energies: Coupling edges in graph cuts. In *CVPR 2011*, June 2011. IEEE, 1897–1904. https://doi.org/10.1109/CVPR.2011.5995589

[35] Stefanie Jegelka and Jeff A. Bilmes. 2017. Graph cuts with interacting edge weights: examples, approximations, and algorithms. *Mathematical Programming* 162, 1–2 (March 2017), 241–282. https://doi.org/10.1007/s10107-016-1038-y

[36] Michael Jungmair, et al. 2022. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[37] Tomas Karnagel, et al. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *Proceedings of the VLDB Endowment* 10, 7 (March 2017), 733–744. https://doi.org/10.14778/3067421.3067423

[38] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, Raymond E. Miller et al. (eds.). Springer US, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[39] Richard A. Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices* 30, 3 (March 1995), 13–22. https://doi.org/10.1145/202530.202532

[40] Timo Kersten, et al. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* 30, 5 (September 2021), 883–905. https://doi.org/10.1007/s00778-020-00643-4

[41] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*, 1973. ACM Press, 194–206. https://doi.org/10.1145/512927.512945

[42] Jon Kleinberg and Éva Tardos. 2002. Approximation algorithms for classification problems with pairwise relationships: metric labeling and markov random fields. *Journal of the ACM* 49, 5 (September 2002), 616–639. https://doi.org/10.1145/585265.585268

[43] P. Kohli, et al. 2009. P^3 & Beyond: Move Making Algorithms for Solving Higher Order Functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 9 (September 2009), 1645–1656. https://doi.org/10.1109/TPAMI.2008.217

[44] V. Kolmogorov. 2006. Convergent Tree-Reweighted Message Passing for Energy Minimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 10 (October 2006), 1568–1583. https://doi.org/10.1109/TPAMI.2006.200

[45] Vladimir Kolmogorov. 2012. Minimizing a sum of submodular functions. *Discrete Applied Mathematics* 160, 15 (October 2012), 2246–2258. https://doi.org/10.1016/j.dam.2012.05.025

[46] Nikos Komodakis, et al. 2008. Performance vs computational efficiency for optimizing single and dynamic MRFs: Setting the state of the art with primal-dual strategies. *Computer Vision and Image Understanding* 112, 1 (October 2008), 14–29. https://doi.org/10.1016/j.cviu.2008.06.007

[47] Sebastian Kruse, et al. 2020. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *The VLDB Journal* 29, 6 (November 2020), 1287–1310. https://doi.org/10.1007/s00778-020-00612-x

[48] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[49] Chris Lattner, et al. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, February 27, 2021. IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[50] Geonho Lee, et al. 2024. Chimera: A System Design of Dual Storage and Traversal-Join Unified Query Processing for SQL/PGQ. *Proceedings of the VLDB Endowment* 18, 2 (October 2024), 279–292. https://doi.org/10.14778/3705829.3705845

[51] Victor Lempitsky, et al. 2010. Fusion Moves for Markov Random Field Optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32, 8 (August 2010), 1392–1405. https://doi.org/10.1109/TPAMI.2009.143

[52] Guy M. Lohman, et al. 1985. Query Processing in R*. In *Query Processing in Database Systems*, Won Kim et al. (eds.). Springer Berlin Heidelberg, 31–47. https://doi.org/10.1007/978-3-642-82375-6_2

[53] Bingqing Lyu, et al. 2025. A Modular Graph-Native Query Optimization Framework. In *Companion of the 2025 International Conference on Management of Data*, June 22, 2025. ACM, 566–579. https://doi.org/10.1145/3722212.3724425

[54] Hongbin Ma, et al. 2016. G-SQL: fast query processing via graph exploration. *Proceedings of the VLDB Endowment* 9, 12 (August 2016), 900–911. https://doi.org/10.14778/2994509.2994510

[55] Alan K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (February 1977), 99–118. https://doi.org/10.1016/0004-3702(77)90007-8

[56] Niko Matsakis. 2016. Introducing MIR. *The Rust Blog*. Retrieved from https://blog.rust-lang.org/2016/04/19/MIR.html

[57] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[58] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[59] Yasuhiko Minamide, et al. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*, 1996. ACM Press, 271–283. https://doi.org/10.1145/237721.237791

[60] Karl J. Ottenstein, et al. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI '90*, 1990. ACM Press, 257–271. https://doi.org/10.1145/93542.93578

[61] Marcus Paradies, et al. 2015. GRAPHITE: an extensible graph traversal framework for relational database management systems. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, June 29, 2015. ACM, 1–12. https://doi.org/10.1145/2791347.2791383

[62] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12, 4–5 (July 2002), 393–434. https://doi.org/10.1017/S0956796802004331

[63] G.D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (December 1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[64] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical Methods in Computer Science* Volume 9, Issue 4, (December 2013), 705. https://doi.org/10.2168/LMCS-9(4:23)2013

[65] The PostgreSQL Contributors. 2026. PostgreSQL Retrieved from https://www.postgresql.org/docs/18

[66] Zhengping Qian, et al. 2021. GAIA: a system for interactive analysis on distributed graphs using a High-Level language. In *18th USENIX symposium on networked systems design and implementation (NSDI 21)*, April 2021. USENIX Association, 321–335. Retrieved from https://www.usenix.org/conference/nsdi21/presentation/qian-zhengping

[67] Karthik Ramachandra, et al. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of VLDB* 11, 4 (December 2017). Retrieved from https://www.microsoft.com/en-us/research/publication/froid-optimization-of-imperative-programs-in-a-relational-database/

[68] Shriram Ramesh, et al. 2021. G r a n i t e : A distributed engine for scalable path queries over temporal property graphs. *Journal of Parallel and Distributed Computing* 151, (May 2021), 94–111. https://doi.org/10.1016/j.jpdc.2021.02.004

[69] Fabrice Rastello and Florent Bouchez Tichadou (Eds.). 2022. *SSA-based compiler design*. Springer International Publishing. Retrieved from https://link.springer.com/10.1007/978-3-030-80515-9

[70] The Rust Project Contributors. 2026. Dataflow Analysis Framework for the Rust Compiler (rustc_mir_dataflow) Retrieved from https://github.com/rust-lang/rust/tree/4cf5f9580233c36f6bc8db76e282ba8a1c1ea491/compiler/rustc_mir_dataflow/src/framework

[71] Amr Sabry. 1998. What is a purely functional language?. *Journal of Functional Programming* 8, 1 (January 1998), 1–22. https://doi.org/10.1017/S0956796897002943

[72] Amr Sabry and Matthias Felleisen. 1992. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, January 1992. ACM, 288–298. https://doi.org/10.1145/141471.141563

[73] Sherif Sakr, et al. 2014. Hybrid query execution engine for large attributed graphs. *Information Systems* 41, (May 2014), 45–73. https://doi.org/10.1016/j.is.2013.10.007

[74] Bernhard Scholz and Erik Eckstein. 2002. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for*

embedded systems: software and compilers for embedded systems, June 19, 2002. ACM, 139–148. https://doi.org/10.1145/513829.513854

[75] Amir Shaikhha, et al. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data*, June 26, 2016. ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

[76] Amir Shaikhha, et al. 2026. Raqlet: Cross-Paradigm Compilation for Recursive Queries

[77] Ruby Y. Tahboub, et al. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data*, May 27, 2018. ACM, 307–322. https://doi.org/10.1145/3183713.3196893

[78] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* 1, 2 (June 1972), 146–160. https://doi.org/10.1137/0201010

[79] Peng Tu and David Padua. 1995. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, June 1995. ACM, 47–55. https://doi.org/10.1145/207110.207115

[80] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

[81] Bobbi W. Yogatama, et al. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

A. OFFSET 0 Materialization Fence: Plan Comparison

A graph read with a single filter body branches on two runtime boolean inputs, producing a nested CASE tree in the generated SQL. This appendix compares the query plans with and without the `OFFSET 0` materialization fence. The fenced plan has a lower estimated cost, but empirical measurement shows that the materialization overhead exceeds the duplication cost for multi-body queries (Chapter 9.3).

A.1. Source Expression

```jexpr
["::graph::tail::collect",
  ["::graph::body::filter",
    ["::graph::head::entities", ["input", "time_axis", "_"]],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      ["if",
        ["input", "foo", "Boolean"],
        ["if",
          ["input", "bar", "Boolean"],
          ["==",
            "vertex.metadata.record_id.entity_id.entity_uuid",
            ["input", "id_a",
            "::graph::types::knowledge::entity::EntityUuid"]
          ],
          ["==",
            "vertex.metadata.record_id.entity_id.entity_uuid",
            ["input", "id_b",
            "::graph::types::knowledge::entity::EntityUuid"]
          ]
        ],
        ["==",
          "vertex.metadata.record_id.entity_id.entity_uuid",
          ["input", "id_c", "::graph::types::knowledge::entity::EntityUuid"]
        ]
      ]
    ]
```

```
    ]
  ]
]
```

A.2. Generated SQL

```sql
SELECT
  ("continuation_2_0"."row")."block" AS "continuation_2_0_block",
  ("continuation_2_0"."row")."locals" AS "continuation_2_0_locals",
  ("continuation_2_0"."row")."values" AS "continuation_2_0_values"
FROM "entity_temporal_metadata" AS "entity_temporal_metadata_0_0_0"
CROSS JOIN LATERAL (
  SELECT
    CASE
      WHEN (($3::jsonb))::int) IS NULL
        THEN
          (ROW(
            COALESCE(((FALSE)::boolean), FALSE),
            NULL,
            NULL,
            NULL
          )::continuation)
      WHEN (($3::jsonb))::int) = 0
        THEN
          (ROW(
            COALESCE(
              (
                (
                  TO_JSONB("entity_temporal_metadata_0_0_0"."entity_uuid")
                  = TO_JSONB(($4::jsonb))
                )::boolean
              ),
              FALSE
            ),
            NULL,
            NULL,
            NULL
          )::continuation)
      WHEN (($3::jsonb))::int) = 1 THEN
        CASE
          WHEN (($5::jsonb))::int) IS NULL
            THEN
              (ROW(
                COALESCE(((FALSE)::boolean), FALSE),
                NULL,
```

```
                  NULL,
                  NULL
                )::continuation)
            WHEN ((($5::jsonb))::int) = 0
              THEN
                (ROW(
                  COALESCE(
                    (
                      (
                        TO_JSONB("entity_temporal_metadata_0_0_0"."entity_uuid")
                        = TO_JSONB(($6::jsonb))
                      )::boolean
                    ),
                    FALSE
                  ),
                  NULL,
                  NULL,
                  NULL
                )::continuation)
            WHEN ((($5::jsonb))::int) = 1 THEN
              (ROW(
                COALESCE(
                  (
                    (
                      TO_JSONB("entity_temporal_metadata_0_0_0"."entity_uuid")
                      = TO_JSONB(($7::jsonb))
                    )::boolean
                  ),
                  FALSE
                ),
                NULL,
                NULL,
                NULL
              )::continuation)
          END
      END AS "row"
      OFFSET 0
) AS "continuation_2_0"
WHERE
  "entity_temporal_metadata_0_0_0"."transaction_time" && ($1::tstzrange)
  AND "entity_temporal_metadata_0_0_0"."decision_time" && ($2::tstzrange)
  AND ("continuation_2_0"."row")."filter" IS NOT FALSE
```

A.3. Parameters

Parameter	Binding
$1	Transaction time axis (tstzrange)
$2	Decision time axis (tstzrange)
$3	Input `foo` (Boolean)
$4	Input `id_c` (EntityUuid)
$5	Input `bar` (Boolean)
$6	Input `id_b` (EntityUuid)
$7	Input `id_a` (EntityUuid)

A.4. Query Plans

Both plans are obtained using `PREPARE` with `SET plan_cache_mode = force_generic_plan` to prevent constant folding of the parameter values.

A.4.1. With `OFFSET 0`

```
Nested Loop  (cost=0.14..8.24 rows=1 width=68)
  → Index Only Scan
      using entity_temporal_metadata_temporal_idx
      on entity_temporal_metadata  (cost=0.14..8.16 rows=1 width=16)
        Index Cond: ((transaction_time && $1)
                 AND (decision_time && $2))
  → Subquery Scan on continuation_2_0  (cost=0.00..0.07 rows=1 width=32)
        Filter: ((continuation_2_0."row").filter IS NOT FALSE)
        → Result  (cost=0.00..0.06 rows=1 width=32)
```

The CASE tree is evaluated once in the Result node. Field access occurs on the materialized result.

A.4.2. Without `OFFSET 0`

```
Index Only Scan
  using entity_temporal_metadata_temporal_idx
  on entity_temporal_metadata  (cost=0.14..8.35 rows=1 width=68)
    Index Cond: ((transaction_time && $1)
             AND (decision_time && $2))
    Filter: ((CASE
      WHEN (($3)::integer IS NULL)
        THEN '(f,,,)'::continuation
      WHEN (($3)::integer = 0)
        THEN ROW(COALESCE(
          (to_jsonb(entity_uuid) = to_jsonb($4)),
```

```
12            false), NULL, NULL, NULL)::continuation
13        WHEN (($3)::integer = 1) THEN CASE
14          WHEN (($5)::integer IS NULL)
15            THEN '(f,,,)'::continuation
16          WHEN (($5)::integer = 0)
17            THEN ROW(COALESCE(
18              (to_jsonb(entity_uuid) = to_jsonb($6)),
19              false), NULL, NULL, NULL)::continuation
20          WHEN (($5)::integer = 1)
21            THEN ROW(COALESCE(
22              (to_jsonb(entity_uuid) = to_jsonb($7)),
23              false), NULL, NULL, NULL)::continuation
24          ELSE NULL::continuation END
25        ELSE NULL::continuation END).filter IS NOT FALSE)
```

The planner has inlined the LATERAL subquery. The full nested CASE tree appears directly in the Filter expression, and each field reference in the SELECT list repeats it.

B. Related Work Synthesis

System	Within-query heterogeneity	Typed IR	Cost-based placement	Capability constraints	Type-aware transfer	Arbitrary k targets
G-SPARQL	✓	×	×	○	×	×
G-SQL	✓	×	○	○	×	×
GOpt	×	✓	×	×	×	○
LingoDB	×	✓	×	×	×	×
Umbra	○	✓	○	×	×	×
LB2	×	✓	×	×	×	×
Rheem	✓	×	✓	○	×	✓
Karnagel	✓	×	✓	×	×	✓
BigDAWG	✓	×	○	○	×	✓
PolyBase	✓	×	○	○	×	×
Mordred	✓	×	○	×	×	×
PRECI*	✓	×	×	✓	×	✓
HashQL	✓	✓	✓	✓	✓	✓
✓ = supported ○ = partial × = not supported						

Table 7: Surveyed systems against the conjunction of properties this work addresses.

C. Benchmark Query Corpus

The following queries constitute the benchmark corpus described in Chapter 9.1. Each query is expressed in J-Expr, the JSON-based surface syntax of HashQL.

```jexpr
["::graph::tail::collect",
  ["::graph::body::filter",
    ["::graph::head::entities", ["input", "temporal_axes", "_"]],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      { "#literal": true },
    ],
  ],
]
```

Listing 21: All Entities: Collect with a constant-true filter.

```jexpr
["::graph::tail::collect",
  ["::graph::body::filter",
    ["::graph::head::entities", ["input", "temporal_axes", "_"]],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      ["==",
        "vertex.metadata.record_id.entity_id.entity_uuid",
        ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
      ],
    ],
  ],
]
```

Listing 22: Filter by UUID: Single equality comparison against an input parameter. No branching.

```jexpr
["::graph::tail::collect",
  ["::graph::body::filter",
    ["::graph::head::entities", ["input", "temporal_axes", "_"]],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      ["if",
        ["==",
          "vertex.metadata.record_id.entity_id.entity_uuid",
          ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
        ],
        ["==",
          "vertex.metadata.record_id.entity_id",
          ["input", "alice_id", "::graph::types::knowledge::entity::EntityId"],
        ],
        ["==",
          "vertex.metadata.record_id.entity_id.entity_uuid",
          ["input", "bob_uuid", "::graph::types::knowledge::entity::EntityUuid"],
        ],
      ],
    ],
  ],
]
```

Listing 23: Diamond CFG: Conditional filter with if-then-else.

```jexpr
["::graph::tail::collect",
  ["::graph::body::filter",
    ["::graph::body::filter",
      ["::graph::head::entities", ["input", "temporal_axes", "_"]],
      ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
        ["if",
          ["!=",
            "vertex.metadata.record_id.entity_id.entity_uuid",
            ["input", "org_uuid", "::graph::types::knowledge::entity::EntityUuid"],
          ],
          ["if",
            ["!=",
              "vertex.metadata.record_id.entity_id.entity_uuid",
              ["input", "draft_alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
            ],
            ["!=",
              "vertex.metadata.record_id.entity_id.entity_uuid",
              ["input", "friend_link_uuid", "::graph::types::knowledge::entity::EntityUuid"],
            ],
            { "#literal": false },
          ],
          { "#literal": false },
        ],
      ],
    ],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      ["==",
        "vertex.metadata.record_id.entity_id.entity_uuid",
        ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
      ],
    ],
  ],
]
```

Listing 24: Sequential Filters: Two chained filters with nested conditionals.

```jexpr
["::graph::tail::collect",
```

```
    ["::graph::body::filter",
      ["::graph::body::filter",
        ["::graph::body::filter",
          ["::graph::head::entities", ["input", "temporal_axes", "_"]],
          ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
            ["||",
              ["&&",
                ["||",
                  ["==",
                    "vertex.metadata.record_id.entity_id.entity_uuid",
                    ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
                  ],
                  ["==",
                    "vertex.metadata.record_id.entity_id.entity_uuid",
                    ["input", "bob_uuid", "::graph::types::knowledge::entity::EntityUuid"],
                  ],
                ],
                { "#literal": true },
              ],
              ["&&",
                ["&&",
                  ["!=",
                    "vertex.metadata.record_id.entity_id.entity_uuid",
                    ["input", "org_uuid", "::graph::types::knowledge::entity::EntityUuid"],
                  ],
                  ["!=",
                    "vertex.metadata.record_id.entity_id.entity_uuid",
                    ["input", "draft_alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
                  ],
                ],
                ["!=",
                  "vertex.metadata.record_id.entity_id.entity_uuid",
                  ["input", "friend_link_uuid", "::graph::types::knowledge::entity::EntityUuid"],
                ],
              ],
            ],
          ],
        ],
        ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
          ["if",
            ["==",
              "vertex.metadata.record_id.entity_id.entity_uuid",
              ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
            ],
            ["==",
              "vertex.metadata.record_id.entity_id",
              ["input", "alice_id", "::graph::types::knowledge::entity::EntityId"],
            ],
            ["if",
              ["==",
                "vertex.metadata.record_id.entity_id.entity_uuid",
                ["input", "bob_uuid", "::graph::types::knowledge::entity::EntityUuid"],
              ],
              ["==",
                "vertex.metadata.record_id.entity_id",
                ["input", "bob_id", "::graph::types::knowledge::entity::EntityId"],
              ],
              { "#literal": false },
            ],
          ],
```

```
      ],
    ],
    ["fn", { "#tuple": [] }, { "#struct": { "vertex": "_" } }, "_",
      ["||",
        ["==",
          "vertex.metadata.record_id.entity_id.entity_uuid",
          ["input", "alice_uuid", "::graph::types::knowledge::entity::EntityUuid"],
        ],
        ["==",
          "vertex.metadata.record_id.entity_id.entity_uuid",
          ["input", "bob_uuid", "::graph::types::knowledge::entity::EntityUuid"],
        ],
      ],
    ],
  ],
]
```

Listing 25: Stress Test: Three sequential filters.