From Categorical Schemas to Physical Plans: Compiling CatDB into Relational, Graph, Document, and Vector Execution

Matthew Long
The YonedaAI Collaboration
YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

23 July 2026

Abstract

A semantic data layer still has to produce physical work. If it takes row execution away from the database engine, it also gives up much of the mature optimizer it was meant to use. This paper specifies a provisional compilation contract for CatDB, a proposed data platform whose schemas and mappings are immutable, typed, versioned artifacts. A bounded canonical query passes through an explicitly directional read mapping and becomes a typed bag-relational intermediate representation. The compiler then marks each backend operation as exact, residual, approximate, or unsupported before it emits a parameterized realization. SQLite and PostgreSQL are the first targets. Graph, document, RDF, vector, and polystore execution remain future preservation boundaries, not implemented backends.

The contract includes an immutable query artifact, a finite-bag denotation, typed path expansion, logical operators, capability manifests, realization certificates, three distinct cache layers, normalized result equivalence, and a lineage comparison. Within the admitted fragment, required scalars and pure total predicates combine with projection, typed path navigation, and inner equijoins. We prove on paper that well-witnessed path expansion preserves types, an exact compositional lowerer preserves the fragment's abstract denotation, and a cached artifact can be reused exactly when all declared semantic and physical dependencies remain unchanged. These results belong to the stated mathematical model. They do not show that a compiler or backend adapter exists.

The implementation evidence is not present. The manuscript reports no CatDB query compiler, query-planner crate, SQLite or PostgreSQL connector, generated SQL artifact, captured native plan, differential result, performance result, Haskell query oracle, or Lean query theorem. The central "outside the hot execution path" claim therefore remains an engineering hypothesis. The evaluation plan is designed to catch semantic mismatches, silent approximations, cache errors, row-dependent

compilation, and unacceptable overhead. The candidate contribution is narrower than a new optimizer. It would preserve versioned semantic intent through a capability-checked handoff, while native engines keep access-path selection, join ordering, and row processing.

Contents

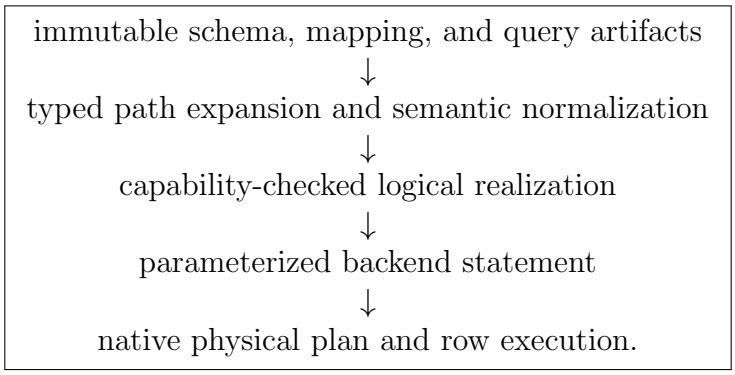
1 Introduction

1.1 Plain-language problem

An analyst asks for customers and their qualifying orders. One source stores the relationship as a foreign key. Another uses a graph edge, while a third nests the order inside a document. A semantic layer can give these structures one typed name, but that name does not execute the query. The compiler still has to decide what the backend preserves, what must run elsewhere, and which requests it cannot answer faithfully. It also has to make those decisions without putting a categorical interpreter inside every row scan.

Database engines already handle logical rewrites, access-path selection, join ordering, pipelining, parallelism, and cost-based choice [19, 8, 9]. Federated systems, mediators, adapters, and ontology-based access already rewrite queries across heterogeneous sources [20, 39, 4, 5]. CatDB should not rename that work and present it as a categorical contribution. The systems question here is smaller: can typed schema mappings provide a stable semantic boundary before a conventional backend optimizer takes control?

The proposed boundary is:



CatDB would own the first three stages. SQLite, PostgreSQL, or another selected engine would own most of the last two. The realization must leave enough evidence to explain that handoff, invalidate caches correctly, and compare the result with an independent evaluator.

1.2 Research question and central hypothesis

The research question is:

> *Can a bounded query over a categorical schema be compiled through versioned mappings into backend-native execution while preserving its declared semantics and keeping semantic compilation outside the per-row hot path?*

The central hypothesis is written so that an implementation can disprove it:

> A bounded CatDB query can be expanded through immutable typed mappings, lowered into a capability-checked logical realization, and emitted as parameterized SQLite or PostgreSQL SQL, with ordinary database operators executing rows and no CatDB semantic callback or row-dependent compilation for a fixed query shape.

The hypothesis does not make compilation free or every query pushable. It does not give SQLite and PostgreSQL identical semantics, and one logical plan need not determine one native physical plan. Graph, document, RDF, and vector semantics also do not become lossless relational operators by assumption.

1.3 Contributions

The provisional specification has seven parts.

1. It defines immutable query, mapping, capability, policy, and realization artifacts, with exact dependency identities rather than a single opaque plan key.
2. It gives a typed finite-bag semantics for a first query fragment: required scalar fields, pure total predicates, projection, path navigation, and inner equijoins.
3. It separates semantic expansion from backend realization and requires each operation to receive one of four dispositions: Exact, Residual, Approximate, or Unsupported.
4. It supplies paper proofs for type preservation, abstract exact-lowering preservation, and dependency-complete cache reuse.
5. It describes separate SQLite and PostgreSQL contracts, including typing, constraints, prepared statements, native-plan evidence, and invalidation boundaries.
6. It provides a preservation matrix for graph, document, RDF, vector, Arrow, Substrait, and polystore boundaries without claiming those backends are implemented.
7. It gives an experiment and reproducibility contract whose failure conditions include semantic mismatch, silent approximation, cache confusion, and per-row compiler work.

1.4 Evidence labels

We use the following labels throughout:

- SUPPORTED BY PRIOR LITERATURE means that a cited primary paper, standard, or official backend document establishes the referenced background result or facility.
- PROVED IN THIS PAPER means that this manuscript contains a complete argument under explicit assumptions. It does not mean machine-checked.
- ENGINEERING HYPOTHESIS means a proposed design or measurable systems claim for which the required implementation evidence does not yet exist.

- OPEN CONJECTURE means a question for which neither this paper nor the current artifacts supply a proof or decisive experiment.
- SPECULATIVE marks a possible extension outside the committed initial profile.

No statement in this paper is labeled machine-checked or experimentally supported.

2 Evidence and current claim boundary

2.1 Evidence cut

This manuscript uses the evidence dossier dated 22 July 2026 as its source cut. That dossier verifies primary literature for query optimization, federation, ontology-based access, cross-model systems, vector search, portable plans, and the official SQLite and PostgreSQL facilities cited below. It also audits the CatDB repository. The audit found a bounded schema and mapping foundation from the preceding semantic-model work, but no P2 execution implementation.

Artifact class	Status at evidence cut	Consequence for this paper
Finite schema and mapping definitions	available from P1	notation may be reused; no P2 execution claim follows
Query profile and finite query evaluator	absent	query semantics here are a paper specification
Rust query or planner crate	absent	no compiled-CatDB claim
SQLite storage connector	absent	SQL examples are proposed realizations
PostgreSQL storage connector	absent	parity and plan claims are untested
Generated SQL and typed parameters	absent	no emission result is reported
Captured backend plan	absent	no native-plan observation is reported
Result and lineage differential	absent	no semantic equivalence is experimentally supported
P2 performance benchmark	absent	no overhead or hot-path number is reported
Haskell query oracle	absent	no cross-language query result is reported
Lean query theorem	absent	no machine-checked P2 theorem is claimed

Table 1: P2 evidence boundary. "Absent" means absent from the audited evidence cut, not impossible or permanently unplanned.

2.2 Claims the evidence does support

Status. SUPPORTED BY PRIOR LITERATURE.

The literature supports several ingredients independently. Cost-based relational optimization and extensible physical operator frameworks are established [19, 8, 9]. Federated and mediated architectures are established [20, 39]; distributed optimization has long had to model shipping and remote costs [12]. Calcite demonstrates a modular framework for heterogeneous query planning [4], while Trino documents connector-specific pushdown [34]. Ontop demonstrates mapping-aware rewriting from an ontology-facing query language to relational sources [5]. These facts motivate the architecture but do not validate its CatDB realization.

SQLite officially documents SQL preparation, byte-code statements, automatic reprepare conditions, explain facilities, per-statement counters, constraints, typing, strict tables, and JSON functions [21, 24, 25, 27, 29, 28, 22, 23, 30, 26]. PostgreSQL 18 officially documents preparation, generic and custom plan behavior, replanning conditions, structured explain output, query-planning settings, data types, and constraints [17, 16, 18, 15, 14]. These are backend capabilities, not measurements of a CatDB adapter.

2.3 Claims deliberately withheld

The paper does not claim:

- that categorical semantics improve SQL runtime;
- that CatDB generates correct SQLite or PostgreSQL SQL today;
- that a warm semantic cache implies a warm backend native-plan cache;
- that explain text is a stable cross-version correctness oracle;
- that PostgreSQL generic and custom plans have equal behavior;
- that SQLite affinity and PostgreSQL scalar types automatically share one equality or collation;
- that every schema equation can be lowered as a native constraint;
- that graph, document, RDF, or vector operators are implemented;
- that an approximate nearest-neighbor result establishes identity, equality, or categorical law; or
- that P1 Rust or Lean evidence proves any P2 execution property.

3 Prior art and candidate delta

3.1 Relational optimization

System R established dynamic-programming access-path and join-order selection as a practical cost-based optimization model [19]. Volcano separated extensible operators and exchange mechanisms [8]; Cascades organized transformation rules and search [9]; Eddies explored runtime adaptation [3]. These systems own the physical choices that CatDB should preserve,

not displace. The proposed CatDB realization ends before native scan, index, join algorithm, parallelism, memory, and scheduling decisions.

The candidate delta is therefore not "categorical cost-based optimization." It is an upstream semantic artifact that records why a logical relationship exists, which versioned mappings expanded it, and which backend features preserve it. Whether that extra boundary is useful at acceptable cost is an ENGINEERING HYPOTHESIS.

3.2 Federation, mediators, and adapters

Federated databases explicitly address distribution, heterogeneity, and autonomy [20]. Mediator architectures place domain-specific processing between applications and data sources [39]. R* studied distributed query costs and shipping choices [12]. Calcite provides a mature logical/physical planning and adapter framework [4]; Trino exposes which operations a connector can push down [34]. BigDAWG uses islands, shims, and casts to coordinate heterogeneous engines [7].

CatDB's proposed distinction is that source-to-domain meaning is an immutable typed mapping with explicit preservation and loss, not only a catalog entry, adapter rule, or cast. This is a representation and governance claim until a compiler demonstrates that the artifacts compose, diagnose changes, and produce competitive plans.

3.3 Ontology-based access and standardized mappings

RDF defines graph and dataset concepts; SPARQL defines graph query semantics; R2RML defines relational-to-RDF mappings [36, 37, 35]. Ontop is a direct prior-art baseline for mapping-aware query rewriting over relational databases [5]. A serious CatDB evaluation must compare mapping expressiveness, rewrite coverage, diagnostics, and execution overhead against this lineage. "Typed mappings" alone are not a novelty claim.

The candidate delta is a single versioned operational record spanning categorical path equations, query expansion, backend dispositions, lineage, and cache dependencies across more than an RDF-to-relational setting. That span is not demonstrated in the present evidence.

3.4 Cross-model systems and portable interchange

AsterixDB is an established semistructured database system [1]; GQL standardizes a property-graph language [10]. Arrow standardizes an in-memory columnar format [2]. Substrait proposes portable compute plans and an extension mechanism [33, 31, 32]. These are distinct layers. Arrow does not by itself specify semantic query intent, and a Substrait extension does not by itself prove that two engines give equivalent results.

CatDB could eventually lower its logical IR to Substrait or exchange record batches through Arrow. Both remain SPECULATIVE in this paper. The initial executable target is a smaller relational subset with explicit SQLite and PostgreSQL contracts.

3.5 Vector execution and proof boundaries

HNSW, Faiss, and Milvus establish approximate nearest-neighbor techniques and vector-system baselines [13, 11, 38]. Approximation changes the result contract: recall, distance metric, index parameters, nondeterminism, and candidate limits become observable semantics. CatDB therefore classifies ANN as Approximate, never silently as Exact. Similarity can generate candidates for later reconciliation; it cannot establish semantic identity or a path equation by itself.

Mechanized SQL semantics such as HoTTSQL show that proof-oriented query rewrite validation is possible for carefully defined languages [6]. This paper's three proofs are conventional mathematics over a much smaller abstract IR. No generated SQL proof or machine-checked compiler theorem is claimed.

4 Compilation architecture

4.1 Four layers and one ownership boundary

Status. ENGINEERING HYPOTHESIS.

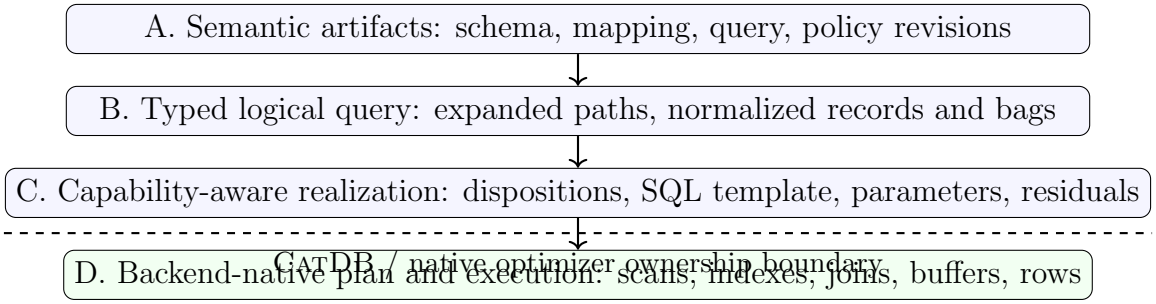


Figure 1: Proposed compilation layers. Layer D is observed and measured but not reconstructed as a categorical evaluator.

Layer A identifies meaning. Layer B has backend-independent typed operators. Layer C chooses a declared backend realization and records every non-exact boundary. Layer D belongs to the native engine. A planner may inspect backend statistics or explain evidence, but it must not pretend to control every physical choice.

4.2 Compile time, prepare time, and row time

The phrase "outside the hot path" is ambiguous unless time is partitioned:

Semantic compile time parses or loads immutable artifacts, validates dependencies, expands mappings, types paths, and constructs logical IR.

Realization time consults the capability manifest, chooses exact/residual dispositions, emits a statement template and typed parameters, and creates lineage rules.

Backend prepare time asks SQLite or PostgreSQL to parse, analyze, rewrite, compile, or plan the statement according to the backend contract.

Row time scans, filters, joins, projects, serializes, and transfers data, plus any explicitly declared residual operator.

The central hypothesis permits fixed query-shape compilation and realization before execution. It permits backend preparation and backend-controlled replanning. It permits a residual operator only when the realization marks it and measures its cost. It is falsified when a fixed query shape invokes the semantic compiler or a semantic callback once per row.

4.3 Failure is an output

Compilation is a total diagnostic process even when realization is partial. For each logical node it returns an exact translation, an explicit residual, an approved approximation, or a stable unsupported diagnostic. A query whose output contract cannot be met has no executable realization. It does not fall through to a best-effort query while retaining an exact label.

5 Mathematical framework

5.1 Schemas, paths, and read mappings

We reuse only the following P1 notation. A finite schema presentation S has entities, typed generating arrows, attributes into a fixed scalar typeside, and equations between parallel entity paths. A typed path $p : A \rightsquigarrow B$ is an endpoint-compatible list of generators. Composition is traversal-order concatenation. These definitions describe a presentation; they do not assert that the presented category is finite or that arbitrary presented equality is decidable.

Query rewriting requires directionality beyond a generic schema functor. A source-to-domain mapping need not be invertible. We therefore do not "run a functor backward" by assumption.

Definition 5.1 (Read-rewrite witness)**.** *Status.* ENGINEERING HYPOTHESIS.

Let T be the query-facing schema and S a source-facing schema. A read-rewrite witness

$$M : T \rightsquigarrow_{\text{read}} S$$

is an immutable artifact that assigns every admitted query-facing entity, attribute, and traversed path a well-typed source expression and records:

1. its source and target schema revisions;
2. the validated mapping revisions from which the assignment was derived;
3. required equations and identity assumptions;
4. totality, cardinality, scalar, and nullability preconditions; and
5. an information disposition for each assignment.

Such a witness may be authored directly, derived from a lawful mapping in a direction that supports reads, or rejected as unavailable. It is not an inverse unless an inverse law is separately established.

5.2 Scalars, records, and finite bags

Definition 5.2 (Query-v1 scalar profile)**.** *Status.* ENGINEERING HYPOTHESIS.

The initial query profile uses required values from a fixed set of scalar domains. Each admitted domain has a canonical codec, a total equality relation, a declared order when comparison is permitted, and a declared collation when textual comparison is permitted. Null, missing, non-finite floating values, implicit coercion, locale-dependent comparison, and volatile functions are outside the profile.

A record type is a finite ordered list $\rho = \langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle$ with distinct labels. Its values form $[\![\rho]\!] = \prod_i D_{\tau_i}$. For a set X, define the finite-bag space

$$\mathcal{B}_{\mathrm{fin}}(X) = \{m : X \to \mathbb{N} \mid \mathrm{supp}(m) \text{ is finite}\}.$$

Bag union adds multiplicities. Every query in the admitted fragment denotes a finite bag of records. No implicit duplicate elimination occurs.

5.3 Expressions and predicates

Scalar expressions are constants, typed parameters, field projections, and admitted typed attributes reached through already-expanded source paths. A predicate is pure and total. Its denotation is a function $[\![\varphi]\!] : [\![\rho]\!] \to \{\mathsf{true}, \mathsf{false}\}$. There is no third truth value in query-v1.

Definition 5.3 (Normalized logical query)**.** *Status.* ENGINEERING HYPOTHESIS.

For a schema S, a normalized logical query is generated by:

$$\begin{aligned} Q \quad ::= \quad & \mathsf{Scan}(A, \alpha) \mid \mathsf{Select}_{\varphi}(Q) \mid \mathsf{Project}_{\pi}(Q) \\ & \mid \mathsf{Join}_{\theta}(Q_1, Q_2) \mid \mathsf{Navigate}_{p,\beta}(Q), \end{aligned}$$

where A is an entity, α, β are aliases, φ is a typed total predicate, π is a typed record projection, θ is a conjunction of equalities between same-domain required fields, and $p : A \rightsquigarrow B$ is a typed path from the input entity.

Navigate is semantic syntax. Normalization replaces it with the scans and inner equijoins selected by a read-rewrite witness and physical storage mapping. The resulting core IR uses only scan, select, project, and inner equijoin.

5.4 Bag denotation

Fix a finite instance and a record encoding for each entity. A scan returns one record per entity occurrence with multiplicity inherited from the physical relation. Selection keeps the multiplicity of records satisfying its predicate. Projection sums multiplicities of input records that map to the same projected record. Inner join multiplies multiplicities for matching pairs:

$$[\![\mathsf{Join}_{\theta}(Q_1, Q_2)]\!]\,(r_1 \bowtie r_2) = [\![Q_1]\!]\,(r_1) \cdot [\![Q_2]\!]\,(r_2)$$

when $\theta(r_1, r_2)$ is true, with multiplicities summed when distinct pairs encode the same output record.

This denotation fixes duplicate behavior. It does not model SQL nulls, outer joins, grouping, ordering, limits, recursion, updates, exceptions, or approximate search.

6 Immutable query artifacts

Definition 6.1 (Immutable query artifact)**.** *Status.* ENGINEERING HYPOTHESIS.

An immutable query artifact is a canonical serialized record

$$\mathcal{Q} = (q, \sigma, \mu, \pi, \Gamma, P, O)$$

where q is a stable query identifier; σ is the query-facing schema revision; μ is the ordered list of read-rewrite and storage mapping revisions; π is the query-profile version; Γ is the ordered typed parameter context; P is normalized semantic syntax; and O is an output contract containing record type, bag semantics, scalar codecs, error behavior, snapshot requirement, and lineage policy.

The artifact is content-addressed after canonical serialization. A mutable name may point to a new artifact, but must not alter the bytes behind an existing digest. Stable identifiers are not substitutes for content digests: both are recorded.

6.1 Parameter and result contracts

Parameter values are not embedded in the query identity. The context $\Gamma = \langle x_1 : \tau_1, \ldots, x_k : \tau_k \rangle$ fixes order, domain, codec, range restrictions, and whether a value class can affect backend plan choice. Runtime values are separately hashed when a result cache or parameter-sensitive physical decision requires them.

The output contract O fixes:

- ordered field labels and scalar domains;
- bag rather than set semantics;
- canonical scalar decoding;
- equality and collation policy;
- snapshot or transaction expectation;
- error-versus-empty-result behavior;
- source-record and transformation lineage granularity; and
- whether result order is semantically irrelevant.

Because query-v1 has no ordering operator, comparison normalizes a result bag independently of physical row order. A driver that happens to return rows in one order supplies no ordering guarantee.

6.2 Rejected artifact states

An artifact is rejected before realization if any path is ill typed, any referenced revision is unavailable, an assignment is directionally ambiguous, a parameter lacks a codec, a predicate is partial or volatile, an output field is duplicated, or the query requests a feature outside its declared profile. Rejection is stable with respect to the same immutable inputs.

7 Running Customer/Order query

7.1 Canonical schema

One query runs through the whole example, so each compilation stage can be checked against the last. Its canonical schema has the entities `Customer`, `Order`, and `Account`; arrows

$$\begin{aligned} \texttt{placedBy} &: \texttt{Order} \to \texttt{Customer}, \\ \texttt{accountOf} &: \texttt{Customer} \to \texttt{Account}, \\ \texttt{billedTo} &: \texttt{Order} \to \texttt{Account}, \end{aligned}$$

and equation

$$\texttt{billedTo} = \texttt{placedBy}; \texttt{accountOf}.$$

The attributes used here are

$$\texttt{customerName} : \texttt{Customer} \to \texttt{string}, \qquad \texttt{orderTotal} : \texttt{Order} \to \texttt{decimal}.$$

The notation comes from P1. This paper does not import any P1 runtime evidence.

7.2 Canonical request

The request takes a required decimal parameter, `minimumTotal`, and returns the ordered record

$$\langle \texttt{customerId}, \texttt{customerName}, \texttt{orderId}, \texttt{orderTotal} \rangle$$

for each order whose `orderTotal` is at least that value. It reaches the customer through `placedBy`. Duplicate rows remain in the result, and the request does not specify an output order.

In semantic syntax:

$$\begin{aligned} Q_{\mathrm{co}} = \mathsf{Project}_{c.id,\ c.customerName,\ o.id,\ o.orderTotal} \\ \left(\mathsf{Select}_{o.orderTotal \geq :minimumTotal} \left(\mathsf{Navigate}_{placedBy,c}(\mathsf{Scan}(\texttt{Order}, o))\right)\right). \end{aligned}$$

7.3 Source realization assumptions

The proposed relational storage mapping uses two tables:

```
catdb_customer(id, name, account_id),
catdb_order(id, customer_id, billed_account_id, total).
```

Every field used by Q_{co} is required. In the mapping witness, `order.customer_id` = `customer.id` realizes `placedBy`. The witness maps `customerName` to `customer.name` and `orderTotal` to `order.total`. The relationship is total only if every visible order has a matching customer in the query snapshot. A disabled or violated foreign key does not establish that fact.

7.4 Normalized logical plan

Expanding the path and normalizing the navigation gives:

$$
\begin{aligned}
J_{\text{co}} &= \mathsf{Join}_{o.customer\_id=c.id}(\mathsf{Scan}(\texttt{catdb\_order}, o), \mathsf{Scan}(\texttt{catdb\_customer}, c)),\\
F_{\text{co}} &= \mathsf{Select}_{o.total \geq :minimumTotal}(J_{\text{co}}),\\
L_{\text{co}} &= \mathsf{Project}_{c.id,\ c.name,\ o.id,\ o.total}(F_{\text{co}}).
\end{aligned}
$$

This logical plan does not prescribe a join algorithm. SQLite can choose its own nested loops and indexes. PostgreSQL can choose nested-loop, hash, or merge strategies from its statistics and optimizer rules. CatDB records the logical equality and observes whichever native plan the engine selects.

7.5 Illustrative SQL templates

The proposed SQLite template is:

```
SELECT c.id AS customer_id,
       c.name AS customer_name,
       o.id AS order_id,
       o.total AS order_total
FROM catdb_order AS o
INNER JOIN catdb_customer AS c
        ON o.customer_id = c.id
WHERE o.total >= ?1
```

PostgreSQL uses the same statement shape with its parameter syntax:

```
SELECT c.id AS customer_id,
       c.name AS customer_name,
       o.id AS order_id,
       o.total AS order_total
FROM catdb_order AS o
INNER JOIN catdb_customer AS c
        ON o.customer_id = c.id
WHERE o.total >= $1
```

No CatDB compiler has emitted or executed either listing. They are test oracles for a future implementation. The parameter has a declared decimal codec and must be bound, not interpolated into the SQL text.

8 Mapping and path expansion

8.1 Expansion algorithm

Status. ENGINEERING HYPOTHESIS.

Let $M : T \rightsquigarrow_{\text{read}} S$ be a read-rewrite witness. Expansion E_M recursively substitutes:

1. query-facing entities with source entity expressions;
2. each traversed generator with its witness path or relationship expression;
3. attributes with typed source expressions;
4. canonical predicates with source-domain predicates only when the scalar codec and comparison law are exact; and
5. output fields with explicit source-to-canonical decoders.

Substitution normalizes adjacent paths by traversal-order concatenation. Every intermediate endpoint is checked. The expansion result includes an ordered derivation whose nodes identify the original query expression, mapping assignment, source expression, type judgment, preservation disposition, and source artifact digest.

8.2 Direction and non-invertibility

If a source-to-domain mapping $F : S \to T$ exists but supplies no read-rewrite witness for a requested T-expression, expansion stops. Object merging, value normalization, object synthesis, and information loss can prevent inverse query rewriting. The compiler must not infer an inverse from naming similarity or from the existence of F.

For composed witnesses $M_1 : T \rightsquigarrow U$ and $M_2 : U \rightsquigarrow S$, the artifact records both revisions and the composed expansion. It also records any strengthening of preconditions. Composition may turn an exact operation into a residual or unsupported one; dispositions form part of the composition result.

8.3 Type preservation

Proposition 8.1 (Typed path expansion)**.** *Status.* PROVED IN THIS PAPER.

Let $M : T \rightsquigarrow_{\text{read}} S$ assign every generator $f : A \to B$ traversed by a query a typed source path

$$M(f) : M(A) \rightsquigarrow M(B).$$

Extend M to paths by mapping the empty path to the empty path and concatenating assigned paths in traversal order. If $T \vdash p : A \rightsquigarrow B$, then

$$S \vdash M(p) : M(A) \rightsquigarrow M(B).$$

Proof. Proceed by induction on the length of p. If p is empty at A, then $M(p)$ is the empty path at $M(A)$, which is typed $M(A) \rightsquigarrow M(A)$.

For the inductive step, write the traversal-order path as $p = p'; f$, where $T \vdash p' : A \rightsquigarrow C$ and $f : C \to B$. By the induction hypothesis, $S \vdash M(p') : M(A) \rightsquigarrow M(C)$. By the witness premise, $S \vdash M(f) : M(C) \rightsquigarrow M(B)$. The target of $M(p')$ equals the source of $M(f)$, so typed path composition gives

$$S \vdash M(p'); M(f) : M(A) \rightsquigarrow M(B).$$

Traversal-order substitution defines this composite to be $M(p)$. □

Remark 8.2 (Scope of the proposition). The proposition proves endpoint typing from a well-formed witness. It does not prove that a witness exists, that it preserves equations, cardinality, null behavior, scalar meaning, multiplicity, or performance. Those are separate validation and realization obligations.

9 Capability manifests and dispositions

9.1 Versioned capability manifest

Definition 9.1 (Capability manifest)**.** *Status.* ENGINEERING HYPOTHESIS.

A capability manifest for backend b is an immutable record

$$C_b = (b, v, \kappa, \mathcal{T}, \mathcal{O}, \mathcal{K}, \mathcal{X}, \mathcal{P}, \mathcal{E})$$

containing:

- backend kind b, exact version v, build or extension fingerprint κ;
- supported scalar types, codecs, comparisons, and collations $\mathcal{T}$;
- supported logical operators and expression fragments $\mathcal{O}$;
- constraint and integrity facilities $\mathcal{K}$;
- transaction, isolation, snapshot, cancellation, and streaming behavior $\mathcal{X}$;
- preparation, parameter, and native-plan policy $\mathcal{P}$; and
- evidence interfaces $\mathcal{E}$, such as structured explain output or stable statement counters.

A manifest is descriptive and testable, not a marketing capability list. Each entry identifies its source: normative backend documentation, connector-level test evidence, or an explicit assumption awaiting a test. Changing a backend library version, compile option, enabled extension, collation, session setting, or connector feature creates a new manifest revision when the change can affect realization or results.

9.2 Four dispositions

Definition 9.2 (Realization disposition)**.** *Status.* ENGINEERING HYPOTHESIS.

For a logical operation n, backend manifest C_b, and execution policy ω, capability checking returns exactly one of:

$\mathsf{Exact}(w)$	backend realization with an exactness witness w;
$\mathsf{Residual}(r)$	declared non-backend operator r with placement;
$\mathsf{Approximate}(a)$	approved approximation contract a;
$\mathsf{Unsupported}(d)$	stable diagnostic d and no executable plan.

An exactness witness identifies the logical node, emitted backend fragment, scalar and multiplicity assumptions, snapshot requirement, and the rule whose denotation is asserted equal. A residual identifies where rows move, which fields are needed, how many rows or bytes are estimated, and which semantics the residual evaluator implements. An approximation identifies the metric, quality contract, parameters, and provenance annotation. An unsupported diagnostic identifies the failed obligation and the minimum capability or rewrite needed.

9.3 Capability checking is conjunctive

An operator is exact only if every relevant dimension is exact. SQL syntax support is insufficient. For the predicate `o.total >= :minimumTotal`, exactness requires the source value, parameter, binding codec, backend comparison, collation if applicable, and result decoding to share the declared scalar semantics. For the navigation join, exactness requires endpoint types, key equality, requiredness, multiplicity, and snapshot behavior. One unsupported child prevents an exact parent unless a declared residual repairs the gap.

Dimension	Exact evidence required	Typical failure
Typing	codec and domain agreement	affinity or implicit coercion changes comparison
Truth	pure two-valued predicate	null introduces unknown
Multiplicity	bag-preserving operator	implicit distinct or object flattening duplicates rows
Navigation	total relationship and matching key	missing target or nonunique join key
Snapshot	declared visibility point	sources read at incomparable times
Error	matching failure behavior	conversion failure becomes null or dropped row
Lineage	complete row transformation rule	residual loses source identifiers

Table 2: Exactness is a conjunction of obligations, not a synonym for "pushdown supported."

Feature	Initial disposition	Reason
Required canonical scalars	exact when codec manifest passes	bounded cross-backend domain
Pure total comparison	exact when scalar/order policy passes	two-valued fragment
Projection	exact	field selection and renaming only
Typed total path navigation	exact through inner equijoin when key contract passes	running vertical slice
Inner equijoin	exact when domains, nullability, and multiplicity agree	admitted bag operator
Parameterized SQL	required	separates query shape from values
Null and missing values	unsupported	three-valued and missing semantics not fixed
Outer joins	unsupported	null introduction not fixed
Aggregation and grouping	unsupported	duplicate and numeric rules not fixed
Order, limit, and ties	unsupported	stable order contract not fixed
Volatile functions	unsupported	purity and repeatability violated
Floating/approximate equality	unsupported	no portable exact domain
ANN vector search	approximate only	recall and metric are observable
Recursive traversal	unsupported	termination and backend parity not fixed
Updates and write propagation	unsupported	bidirectional semantics belong to P8
Cross-source transaction	unsupported	no atomic snapshot contract

Table 3: Admitted and excluded first-slice features.

9.4 Profile-v1 capability matrix

10 Logical lowering and preservation

10.1 Core target algebra

The abstract target algebra has bag scans, selection, projection, and inner equijoin with the same typed scalar functions as Section 5. A backend realization may serialize these operators as SQL, but the theorem in this section concerns an abstract target denotation, not a specific SQL implementation.

Definition 10.1 (Exact lowerer)**.** *Status.* ENGINEERING HYPOTHESIS.

An exact lowerer L for the normalized core fragment is defined compositionally:

$$\begin{aligned} L(\mathsf{Scan}(R,\alpha)) &= \mathsf{BScan}(R,\alpha),\\ L(\mathsf{Select}_{\varphi}(Q)) &= \mathsf{BSelect}_{L(\varphi)}(L(Q)),\\ L(\mathsf{Project}_{\pi}(Q)) &= \mathsf{BProject}_{L(\pi)}(L(Q)),\\ L(\mathsf{Join}_{\theta}(Q_1,Q_2)) &= \mathsf{BJoin}_{L(\theta)}(L(Q_1),L(Q_2)). \end{aligned}$$

For every scan, expression, predicate, projection, and equality, the realization certificate states that its target interpretation equals the logical interpretation under the declared record encoding.

10.2 Exact lowering theorem

Proposition 10.2 (Denotation preservation for the admitted core)**.** *Status.* PROVED IN THIS PAPER.

Let Q be a well-typed finite query generated by scan, select, project, and inner equijoin. Suppose:

1. *source scans are finite and their target record encodings preserve each record and multiplicity;*
2. *every lowered scalar expression, predicate, projection, and equality has the same total denotation as its logical source;*
3. *both source and target use the finite-bag definitions in Section 5; and*
4. *execution observes the same source snapshot.*

Then the abstract exact lowerer preserves denotation:

$$[\![L(Q)]\!]_{\text{target}} = [\![Q]\!]_{\text{logical}}\,.$$

Proof. Proceed by structural induction on Q.

Scan. For $Q = \mathsf{Scan}(R,\alpha)$, premise 1 says that $\mathsf{BScan}(R,\alpha)$ returns exactly the encoded logical records with the same finite multiplicities at the same snapshot. Thus the two bags are equal.

Selection. Let $Q = \mathsf{Select}_{\varphi}(Q_0)$. By the induction hypothesis, $[\![L(Q_0)]\!] = [\![Q_0]\!]$. For each record r, premise 2 gives $[\![L(\varphi)]\!](r) = [\![\varphi]\!](r)$. Logical and target selection therefore either retain or discard the same record. When retained, each uses the input multiplicity; hence the output multiplicities agree pointwise.

Projection. Let $Q = \mathsf{Project}_{\pi}(Q_0)$. By induction the input bags agree, and by premise 2 the projections map each input record to the same output record. For any output s, both semantics assign the sum of multiplicities of all input records r for which $\pi(r) = s$. The summation index and every summand agree, so output multiplicities agree.

Inner equijoin. Let $Q = \mathsf{Join}_{\theta}(Q_1,Q_2)$. By induction the left and right input bags agree. Premise 2 gives the same truth value for $\theta(r_1,r_2)$ on every pair. For a matching pair, both semantics contribute the product of the two input multiplicities to the merged output record;

for a nonmatching pair, both contribute zero. Both sum contributions when more than one pair encodes the same output. Therefore every output multiplicity agrees.

These cases exhaust the core grammar. Premise 4 rules out differences caused by reading different snapshots. Hence the finite result bags are equal. □

Remark 10.3 (What the theorem does not prove). The theorem is conditional on exact primitive realizations. It does not show that SQLite SQL, PostgreSQL SQL, a driver codec, a connector, or emitted lineage satisfies those premises. Establishing those facts requires the differential and backend tests in Section 17. A future machine-checked development would also need an explicit SQL semantics or a verified refinement to the driver-visible subset.

10.3 Realization certificate

Definition 10.4 (Realization certificate)**.** *Status.* ENGINEERING HYPOTHESIS.

An executable realization is paired with a canonical certificate

$$\mathcal{R} = (h_Q, h_S, h_M, h_C, h_\omega, L, D, \mathrm{stmt}, \Gamma_b, O_b, \Lambda),$$

where the hashes identify the query, schemas, mappings, capability manifest, and policy; L is normalized logical IR; D assigns dispositions to every node; stmt is the parameterized backend template; Γ_b is the backend binding layout; O_b is the backend decoding contract; and Λ is the lineage derivation.

The certificate is evidence about what the compiler intended. It is not a certificate of backend correctness until tests establish the primitive premises and bind the evidence to an exact backend and connector version.

10.4 Residual plans

A residual plan splits L at an explicit boundary. The backend subplan projects every field required by the residual and lineage evaluator. The certificate records transfer schema, predicted rows and bytes, ordering assumptions, memory policy, and failure semantics. Result comparison covers the complete backend-plus-residual composition. A residual is not a failure by itself; an undeclared residual is.

11 SQLite realization contract

11.1 Why SQLite is the first executable target

SQLite provides a compact embedded environment, a documented preparation API, prepared-statement counters, explain interfaces, ordinary indexes and joins, foreign keys, strict tables, and a controllable file-backed fixture [21, 27, 29, 28, 30]. That makes it useful for the first vertical slice. It does not make SQLite semantically simple: dynamic typing, affinity, collation, foreign-key configuration, and explain instability all affect the contract [23, 24, 25].

11.2 Pinned manifest

The SQLite capability manifest must record at least:

- exact library version, source identifier when available, compile options, binding crate version, and connector revision;
- database encoding, collation set, strict-table policy, and JSON extension assumptions;
- foreign-key enforcement state for every connection;
- transaction and snapshot mode;
- allowed scalar storage classes and canonical codecs;
- parameter binding API and error mapping;
- explain and statement-counter interfaces used as evidence; and
- reprepare behavior visible through counters.

SQLite foreign keys are configured per connection, so a schema declaration alone does not prove total navigation [28]. The acceptance harness must query the live setting, attempt violating writes in an isolated fixture, and record the outcome.

Definition 11.1 (Portable decimal-v1 codec)**.** The initial portable scalar profile admits $\mathsf{Decimal64}(p, s)$ only when

$$0 \le s \le p \le 18, \qquad v = m\,10^{-s}, \qquad |m| < 10^{p},$$

with signed integer mantissa m. The canonical bytes encode the profile revision, p, s, and the two's-complement signed 64-bit mantissa in one declared byte order. SQLite stores and binds m as `INTEGER`; the scale belongs to the immutable type, not to row text. Comparison pushdown is exact only for equal (p, s) domains. A conversion that changes scale must prove exact divisibility and absence of overflow, otherwise it is residual or unsupported. PostgreSQL `numeric` values enter the portable subset only when their decoded value round-trips to the same bounded mantissa. Text collation is never used for decimal order.

11.3 Type and constraint lowering

The initial profile should use `STRICT` tables when the pinned SQLite version and connector support them, while still validating exact codec behavior [30]. SQLite's storage classes and affinity rules remain relevant; a declared decimal domain cannot be assumed exact merely because a column name or declaration contains "decimal" [23]. The first vertical slice uses the bounded $\mathsf{Decimal64}(p, s)$ profile above and tests its round trip, range, ordering, parameter binding, and rejection boundaries. Wider precision and mixed-scale arithmetic remain unsupported until they receive a separate profile.

Primary keys, unique constraints, not-null constraints, checks, and foreign keys are available but have specific documented behavior [22, 28]. A path equation spanning multiple relationships generally is not lowered by pretending one row-local check can enforce it. It becomes a validation query, trigger-backed engineering choice, or application obligation, with its discharge site recorded.

11.4 Preparation and evidence

SQLite's prepare interfaces compile SQL into a prepared byte-code statement [21]. The connector must distinguish:

1. a warm semantic compilation cache;
2. a warm CatDB realization cache containing the SQL template and binding layout; and
3. a live prepared SQLite statement on a particular connection.

The experiment logs preparation count, reprepare count, full-scan steps, sorts, automatic indexes, virtual-machine steps, and statement memory where the official counters expose them [27, 29]. These metrics diagnose physical behavior; they do not prove result correctness.

SQLite warns that both raw `EXPLAIN` and `EXPLAIN QUERY PLAN` output can change between releases [24, 25]. Tests may archive the raw output and extract version-scoped diagnostics, but must not use cross-version text matching as a semantic oracle. Stable correctness assertions compare normalized results, errors, and lineage.

11.5 Running-query acceptance

For Q_{co}, a future SQLite acceptance bundle must contain:

- canonical schema, mapping, query, capability, and policy artifacts;
- fixture DDL, data, indexes, pragma state, and snapshot boundary;
- normalized logical IR and node dispositions;
- the exact SQL bytes and ordered parameter binding;
- an independent finite-evaluator result and lineage bag;
- decoded SQLite result and lineage bag;
- equality comparison, including duplicate cases;
- raw explain evidence and stable statement counters; and
- cold, warm-realization, and warm-prepared latency distributions.

Until this bundle exists, the SQL in Section 7 remains an illustrative target.

12 PostgreSQL realization contract

12.1 Portable subset before backend extensions

PostgreSQL is the second target. Its initial purpose is not to showcase every backend feature, but to test the same declared query-v1 subset under a different type, transaction, planner, and driver environment. A query can be portable even when native physical plans differ. The parity contract compares logical results, errors, scalar values, and lineage, not identical join algorithms.

12.2 Pinned manifest

The PostgreSQL manifest records:

- exact server major and minor version, driver and connector versions, enabled extensions, database encoding, locale, and collations;
- table and column types, domains, constraints, indexes, and statistics snapshot;
- transaction isolation, read-only state, and snapshot procedure;
- session planner settings and resource settings relevant to the run;
- prepared-statement policy, including automatic, generic, or custom behavior;
- binary or text parameter/result codecs and their versioned tests; and
- exact `EXPLAIN` options and permission boundary.

PostgreSQL data types are richer than the initial CatDB scalar profile [15]. The connector must select an exact representation; it must not equate similarly named source and canonical types without round-trip, comparison, overflow, and error tests.

12.3 Preparation and native planning

PostgreSQL `PREPARE` separates parse, analysis, and rewrite work from later execution, while planning may use generic or custom plans and may change after relevant definition or statistics changes [17]. Therefore a CatDB realization key does not name one permanent PostgreSQL physical plan. The evidence record identifies the plan policy and parameter class. Experiments report generic, custom, and automatic modes when skewed parameters could matter.

`EXPLAIN` can expose plan structure, estimates, actual rows and time, buffers, settings, memory, serialization, and other details depending on options and privileges [16]. `ANALYZE` executes the statement, so tests use isolated read fixtures or a rollback-safe protocol. Explain evidence is stored as structured JSON where supported, together with the exact command, settings, and server version.

12.4 Constraints and semantic obligations

PostgreSQL supports not-null, check, unique, primary-key, exclusion, and foreign-key constraints, but their documented semantics set limits [14]. A check condition is satisfied by true or null, which differs from query-v1's required two-valued predicate unless not-null evidence also exists. A check is not a sound mechanism for arbitrary cross-row invariants. A foreign key can support a total-navigation claim only when column nullability, key uniqueness, constraint validation, transaction visibility, and any deferral policy all match the witness.

12.5 Running-query parity

The PostgreSQL vertical slice reuses Q_{co}, the same canonical fixtures, output contract, and oracle result as SQLite. Backend DDL and scalar codecs may differ, but the normalized

result bag and lineage bag must match. The bundle additionally records estimated and actual cardinalities, buffers, planning and execution times, plan-cache mode, and planner settings. A difference is classified as:

1. a connector or compiler defect;
2. an undeclared capability difference;
3. an invalid portable-profile assumption;
4. an oracle defect; or
5. an expected native physical difference with equal semantics.

No category is assigned without a minimal reproducer and preserved evidence.

13 Constraints, residuals, and failure semantics

13.1 Three discharge sites

Status. ENGINEERING HYPOTHESIS.

Every semantic obligation has a meaning and a discharge site. The initial sites are:

Static path typing, artifact totality, output typing, and syntactic profile membership are checked before backend access.

Native a backend constraint or operator discharges the obligation under a pinned manifest and live configuration.

Validation or residual an explicit query, incremental checker, or local operator produces evidence when no native mechanism exactly matches.

A higher-level invariant may remain unresolved. The compiler reports it rather than converting "not natively enforced" into "false" or "unimportant."

13.2 The account path equation

The running schema declares `billedTo` = `placedBy`; `accountOf`. With relational columns, the equality can be tested by joining orders, customers, and accounts and comparing the two account identifiers. That validation query is different from a static categorical derivation and different from a native row-local constraint. Its result is snapshot-scoped: success on one snapshot does not prove all future writes preserve the law.

If Q_{co} traverses only `placedBy`, the account equation need not be evaluated per row merely because it exists in the schema. Its validation state is a dependency only if the selected mapping, policy, or query relies on the equation. This selective obligation graph is important to keeping unrelated semantics out of the row path.

13.3 Partial source availability

The initial single-source relational slice fails closed when its required source or snapshot is unavailable. A future federated profile may allow a partial result only if the query artifact declares a partial-answer semantics, the result carries missing-source provenance, and the consumer accepts it. Returning an incomplete bag under the exact result contract is a semantic mismatch.

13.4 Error normalization

The result comparator treats errors as typed outcomes. It distinguishes at least artifact rejection, unsupported capability, parameter-codec failure, source connection failure, backend statement failure, cancellation, timeout, snapshot failure, residual failure, and result-decoding failure. Backends need not use the same raw error text. Connectors map documented cases to a versioned canonical class and retain the raw diagnostic as evidence.

13.5 Information loss

Projection intentionally loses fields but preserves the declared output contract. Other loss is more consequential. Flattening a document can lose missing-versus-null distinctions; graph lowering can lose edge identity; relational-to-RDF mapping can alter term construction; vector search is approximate; cross-model casts can lose ordering or multiplicity. Each such boundary receives an explicit loss record. An exact label is unavailable when the loss can affect the query's result contract.

14 Plan caches, dependencies, and invalidation

14.1 Three caches, not one

Status. ENGINEERING HYPOTHESIS.

CatDB distinguishes:

1. the *semantic compilation cache*, which stores typed expansion and logical IR for immutable semantic artifacts;
2. the *realization cache*, which stores capability-checked dispositions, a backend statement template, bindings, decoders, residuals, and lineage rules; and
3. the *backend statement or native-plan cache*, which is scoped to a SQLite connection or PostgreSQL session and remains partly backend-controlled.

"Warm plan" is otherwise too imprecise to reproduce. A benchmark labels each layer independently and records whether backend preparation, reprepare, or replanning occurred.

14.2 Dependency-complete keys

Definition 14.1 (Semantic compilation key)**.** The semantic cache key is

$$K_s = H(q, \sigma, \mu, \pi, \nu),$$

where q is the canonical query digest, σ the ordered digests of the query's validated schema dependency closure, μ the ordered mapping and rewrite-witness dependency digests, π the query-profile revision, and ν the semantic compiler implementation revision. The closure names every referenced object, arrow, equation, scalar codec, constraint, and validation witness. The artifact retains the full schema revisions for provenance, but a new revision may reuse K_s only when a validator emits a compatibility witness proving that this complete referenced closure is unchanged.

Definition 14.2 (Realization key)**.** The realization key is

$$K_r = H(K_s, b, C_b, \omega, \lambda, \chi),$$

where b identifies the backend family, C_b is the complete capability manifest digest, ω is the execution-policy digest, λ is the storage-layout and statistics-class digest, and χ identifies the lowerer and connector implementation revisions.

The key does not generally include parameter values, because a statement template is parameterized. If a CatDB policy makes a value-sensitive realization choice, it includes a declared parameter class. PostgreSQL may still select a custom plan for a particular execution, and SQLite may automatically reprepare under documented conditions. Those native events are evidence attached to an execution, not proof that K_r was incorrectly formed.

14.3 Invalidation events

The following events can invalidate K_s: query changes, a change inside the validated schema or mapping dependency closure, rewrite-witness changes, scalar-profile changes, or semantic compiler revision changes. An unrelated schema edit does not force a global flush, but reuse under the new revision is forbidden until the dependency validator emits the compatibility witness. The following can invalidate K_r without invalidating K_s: backend version or build changes, extension changes, collation changes, storage mapping or index-policy changes, connector changes, execution-policy changes, or a statistics-class decision used by the CatDB planner.

Backend caches have additional lifetimes. Closing a SQLite connection loses its prepared statement. PostgreSQL sessions and prepared statements have server-defined replanning and invalidation behavior [17]. SQLite may automatically recompile a prepared statement [21]. CatDB logs these events when observable but does not redefine them.

14.4 Cache-soundness proposition

Proposition 14.3 (Dependency-complete cache reuse)**.** *Status.* PROVED IN THIS PAPER.

Let a deterministic compiler C take the dependency tuple

$$d = (q, \sigma, \mu, \pi, \nu, b, C_b, \omega, \lambda, \chi)$$

and return either a stable diagnostic or a realization artifact. Let $K = H(\mathrm{canon}(d))$, where canonical serialization is injective over admitted dependency tuples and H is collision-free for the tuples under consideration. Consider a cache protocol that authorizes reuse exactly when keys are equal and otherwise recomputes or explicitly revalidates. The protocol authorizes reuse exactly when all declared dependencies in d_0 and d_1 are unchanged, and every reuse it authorizes is sound.

Proof. First suppose every declared dependency is unchanged. Then $\mathrm{canon}(d_0) = \mathrm{canon}(d_1)$, so the keys agree. Determinism gives $C(d_0) = C(d_1)$. Returning the cached diagnostic or realization therefore returns the same artifact the compiler would produce, which is sound.

Conversely, suppose at least one declared dependency changes. Injectivity of canonical serialization gives $\mathrm{canon}(d_0) \neq \mathrm{canon}(d_1)$. Collision freedom on the admitted tuples gives $K_0 \neq K_1$. The artifact stored under K_0 therefore cannot be returned by an authorized successful lookup for K_1. Reusing it by ignoring the changed dependency would violate the cache protocol and is not established sound by determinism, because the compiler is permitted to depend on every declared component.

Thus the protocol authorizes reuse exactly under equality of all declared dependencies, each authorized reuse returns the deterministic compiler result, and any declared change requires a miss or explicit revalidation. □

Remark 14.4 (Assumptions and practical hashing). The proposition states an ideal key contract. Real implementations use finite cryptographic digests, so collision resistance replaces literal collision freedom and canonicalization tests become part of the acceptance suite. The proposition also does not say that the declared dependency set is complete. Tests must mutate each candidate dependency and verify the expected hit, miss, or revalidation transition.

14.5 Negative-cache entries

Stable unsupported diagnostics may be cached under the same dependency discipline. This avoids repeating expensive validation for a known unsupported query without converting the diagnostic into permanent policy. A new capability manifest, mapping, or compiler revision creates a new key and can make the query realizable.

15 Cross-model preservation boundaries

15.1 Why a relational first slice is not a universal model

The title names relational, graph, document, and vector execution because the research program must state how their boundaries differ. It does not imply that one completed compiler currently targets all four. The only proposed first executable slice is relational SQLite, followed by a PostgreSQL portable subset.

15.2 Property graphs

GQL gives a standardized property-graph language boundary [10]. A CatDB entity may correspond to a vertex label and a generator to an edge label, but the correspondence is not automatic. A graph can distinguish parallel edge identity, attach properties to edges, and assign multiplicity to paths in ways that a total functional arrow does not preserve. Variable-length traversal adds termination and duplicate questions outside query-v1.

A future graph capability manifest must state whether a relationship is functional, whether endpoint and edge identity are both returned, how path walks are counted, how labels and properties are typed, and how graph snapshots are obtained. A single-hop functional relationship may lower exactly under those conditions. General graph pattern matching remains unsupported until its semantics enter the profile.

15.3 RDF and ontology-mediated access

RDF graphs, SPARQL solutions, R2RML term maps, and entailment regimes have their own normative semantics [36, 37, 35]. CatDB must not equate a closed finite total instance with an open RDF graph. A future mapping has to record IRI and literal construction, blank-node scope, named-graph selection, entailment assumptions, and multiplicity behavior.

Ontop is a necessary comparative baseline because it already answers SPARQL queries through mappings over relational databases [5]. A CatDB study should use equivalent source data and query families, then compare expressiveness, rewrite time, SQL, result semantics, mapping-change diagnostics, and provenance. Without that comparison, claims of a categorical advantage over ontology-based data access would be premature.

15.4 Documents

Document systems can preserve nested structure that a flat relational view must reconstruct. SQLite JSON functions make JSON queryable, but SQLite does not acquire a new native JSON storage class merely because those functions exist [26]. A document capability manifest must distinguish absent fields from explicit null, arrays from bags, stable array position from unspecified result order, scalar heterogeneity from one typed domain, and duplicate-key parser policy.

For a bounded, required, homogeneous nested field, a path can be exact. Unnesting an array multiplies rows and requires an explicit bag rule. Flattening an object can lose containment or order. Those decisions belong in the realization certificate and result oracle.

15.5 Vectors

Vector execution is different because the common operator is approximate. HNSW and large-scale GPU similarity search establish the performance motivation and tradeoffs [13, 11]; Milvus supplies a purpose-built vector-system baseline [38]. A vector capability manifest must pin dimension, element type, normalization, distance metric, index family, construction parameters, search parameters, filters, tie policy, and index revision.

The result artifact records that candidates were generated approximately. It carries distance or score, source vector and model provenance, index revision, and measured quality profile. No threshold converts a probable match into canonical identity without a separate reconciliation decision.

15.6 Arrow, Substrait, and polystore composition

Arrow may reduce serialization costs at process boundaries and standardize columnar buffers [2]. Substrait may provide a portable relational-plan encoding and an extension mechanism [33, 32]. BigDAWG shows that heterogeneous execution also requires explicit engine islands, shims, and casts [7]. These are complementary candidates: CatDB could provide semantic identity and obligations, Substrait a compute plan, Arrow data exchange, and a polystore layer engine routing.

The experiment must nevertheless measure whether adding an interchange layer helps or merely adds conversion overhead for embedded SQLite. Initial SQLite execution may be simpler through a direct connector. Arrow and Substrait are not architectural requirements until an acceptance test justifies them.

16 Normalized results and lineage

16.1 Result normalization

Definition 16.1 (Normalized result equivalence)**.** *Status.* ENGINEERING HYPOTHESIS.

For query artifact $\mathcal{Q}$ with output record type ρ, a backend result normalizer:

1. decodes each field through the declared canonical scalar codec;
2. rejects extra, missing, null, malformed, or out-of-range fields;
3. orders fields by the output contract;
4. forms a canonical byte representation of each record; and
5. counts equal record encodings to produce a finite bag.

Two successful executions are equivalent when these bags are identical. Two failed executions are equivalent when their canonical error classes and required evidence agree.

Sorting may be used as a comparison implementation, but it does not add an ordering guarantee to the query. Decimal and date values are compared after canonical decoding, not as untyped driver strings. Duplicate-sensitive fixtures are mandatory because set comparison would miss join and projection errors.

16.2 Lineage atoms and expressions

For the initial single-source slice, a lineage atom is

$$\ell = (\text{source revision}, \text{entity}, \text{stable row key}).$$

A query result row carries a finite provenance expression built from atoms, mapping and query revision identifiers, and the transformations that combined or projected them. Scan introduces an atom; selection retains its input lineage; projection retains the contributing lineage; inner join combines the left and right evidence. A residual or approximation adds an explicit transformation node.

The initial comparison need not solve arbitrary provenance equivalence. It uses a canonical normalized lineage form for the bounded operator fragment. The form must distinguish two duplicate result rows with different sources even when their visible values coincide.

Definition 16.2 (Lineage mode)**.** Each query result contract selects one versioned mode:

RequiredAtoms, RequiredNormalized, NoneAllowed.

The first two require the lowerer to preserve exactly the columns and transformations needed by the declared lineage contract. NoneAllowed may prune lineage work only when the request, policy, audit, and comparison contract do not require it. A connector cannot silently downgrade a required mode. The decision and resulting projection are recorded in the realization certificate.

16.3 Why values alone are insufficient

Two plans can produce equal visible values while drawing from different source records, using different mappings, or omitting evidence. Such a difference matters when users ask whether a result survives a schema change or which source version contributed. P2 therefore tests both value bags and lineage bags. Full semiring provenance belongs to the broader P13 program; this paper requires only enough structure to falsify the first compiler slice.

17 Experimental evaluation plan

17.1 Principles

No experiment in this section has been run. The plan is part of the ENGINEERING HYPOTHESIS acceptance contract. A valid report preserves failed outputs, records exact revisions, separates compile, realization, preparation, and row time, and publishes raw observations with the analysis.

Every run records:

- repository commit and dirty-state manifest;
- compiler, connector, Haskell, and Lean revisions when present;
- schema, mapping, query, policy, capability, and fixture digests;
- hardware, operating system, process limits, backend versions and build options;
- DDL, data-generation seed, statistics procedure, indexes, and source snapshot;
- SQL template, ordered parameters, normalized IR, dispositions, residuals, and lineage rules;

- warm/cold state of all three caches;
- latency distributions rather than one timing;
- rows and bytes at every material boundary;
- backend call counts and prepare/replan/reprepare observations;
- native explain or counter evidence; and
- normalized results, lineage, errors, and comparison verdicts.

17.2 Experiment matrix

Table 5: Planned P2 experiments.

ID	Question	Required evidence	Falsification condition
`P2-E01`	Does parsing, expansion, and normalization produce one typed IR?	fixtures for identity, composition, valid and invalid paths; canonical IR and diagnostics	nondeterminism, ill-typed accepted path, or unexplained mismatch
`P2-E02`	Does an independent finite evaluator implement the declared bag semantics?	hand-computed duplicate-sensitive cases and property tests	disagreement with hand oracle or unstable result
`P2-E03`	Does SQLite match the oracle?	emitted SQL and parameters, decoded values, lineage, errors, SQLite manifest	any unclassified value, bag, error, or lineage mismatch
`P2-E04`	Does PostgreSQL match the portable subset?	same canonical cases for each pinned major, with PostgreSQL manifest	any unclassified cross-backend or oracle mismatch
`P2-E05`	Are unsupported features rejected?	negative fixtures for null, outer join, aggregation, order/limit, volatile calls, recursion, updates	executable exact plan or unstable diagnostic
`P2-E06`	Are residuals visible and correct?	forced non-pushdown case, transfer schema, rows/bytes, residual trace, oracle comparison	hidden local work or semantic mismatch
`P2-E07`	Is approximation labeled?	vector-candidate fixture, metric/index parameters, quality and provenance record	result labeled exact or identity inferred automatically
`P2-E08`	Are semantic cache transitions correct?	one-at-a-time mutation of query, schema, mapping, profile, compiler revision	stale hit or unnecessary miss outside stated policy

ID	Question	Required evidence	Falsification condition
P2-E09	Are realization cache transitions correct?	mutations of backend, capability, policy, layout, lowerer, connector	stale realization or conflated cache layer
P2-E10	Does the semantic layer stay outside row time?	stack/profile traces and call counts across growing row counts with fixed shape	compiler/semantic callback count grows with rows
P2-E11	Are parameter-sensitive plans measured honestly?	skewed values; PostgreSQL auto/generic/custom; SQLite reprepare counters	values hidden, one mode generalized, or events conflated
P2-E12	Is compiled execution competitive with a fair native baseline?	equivalent parameterized native SQL, cold/warm distributions, confidence intervals	declared noninferiority threshold fails
P2-E13	Does capability rejection catch scalar and collation mismatches?	adversarial Unicode, decimal, range, comparison, and codec fixtures	silent coercion or different normalized result
P2-E14	Does plan evidence explain material choices?	estimates, actuals, buffers/counters, settings, row/byte flows	plan claim without versioned native evidence
P2-E15	Can another environment reproduce the result?	clean checkout protocol, locked fixtures, machine-readable summary, checksums	documented run cannot be repeated or evidence is missing

17.3 Semantic compilation benchmarks

The compilation suite varies entities, generators, path length, query nodes, mapping-chain length, and rejected obligations independently. It measures schema loading only when not already cached, path expansion, type checking, logical normalization, capability checking, SQL emission, and artifact serialization separately. The existing program target of less than 100 ms for a 100-object schema is a target, not a result.

Mapping validation and query expansion are separated. A benchmark that revalidates all mappings on every query execution measures a different architecture from one that consumes immutable validated artifacts. Both may be useful, but their cache state must be named.

17.4 Execution baselines

The fair native baseline for Q_{co} is a human-reviewed parameterized SQL statement with the same schema, data, indexes, transaction, parameter values, output fields, bag behavior, and driver decoding. It is not an unparameterized literal query, a different projection, or a statement run under different cache conditions.

At minimum, compare:

1. native prepare plus execute;
2. cold CatDB compile, realize, prepare, and execute;
3. warm semantic compile but cold realization and prepare;
4. warm semantic and realization caches but cold backend prepare;
5. warm semantic and realization caches with a reusable prepared statement; and
6. a forced residual plan.

The program's provisional noninferiority target is less than ten percent median overhead for cached execution against the fair native baseline. A warm realization lookup is provisionally expected to be at least ten times faster than uncached physical realization. These thresholds are pre-registered targets only. Failing them revises the claim; it is not filtered out as an inconvenient run.

17.5 Datasets and selectivity

The running fixture includes:

- one customer with multiple qualifying orders;
- distinct customers with equal visible names;
- orders exactly at, below, and above the decimal threshold;
- duplicate visible projections with distinct stable row keys;
- boundary scalar values and adversarial text;
- invalid relationship data in a fixture where enforcement is intentionally disabled; and
- enough scale and skew to change plausible native plan choices.

Scale points should include tiny correctness fixtures, medium in-memory fixtures, and larger fixtures that exceed relevant caches. PostgreSQL statistics are refreshed under a recorded procedure. SQLite indexes and pragma settings are identical across comparable runs.

17.6 Cost and cardinality evidence

For each logical and physical boundary, report estimated and actual rows, estimated and actual transferred bytes where measurable, source calls, and selectivity. PostgreSQL structured explain evidence can expose estimates, actuals, buffers, settings, and other metrics [16]. SQLite counters expose different information [29]; missing metrics are reported as unavailable rather than estimated from undocumented plan text.

A later cost model can include source latency, transfer volume, join cardinality, remote compute price, freshness, transformation cost, provenance-retention cost, consistency, and source load. P2 only constructs the evidence needed to test a first planner boundary.

17.7 Hot-path falsification

The central systems claim fails under any of the following:

1. a semantic compiler or mapping interpreter is invoked once per input or output row for a fixed query shape;
2. a row value triggers row-dependent semantic recompilation rather than declared backend planning or a bounded parameter-class choice;
3. an unsupported operation executes through an unrecorded fallback;
4. an approximate operator is reported as exact;
5. normalized value or lineage equivalence fails;
6. cache reuse ignores a changed declared dependency;
7. the prepared statement or native-plan state is misreported; or
8. the pre-registered performance threshold fails on the stated workload.

Failure of one backend does not establish that the architecture is impossible, but it does falsify the associated backend and workload claim. The manuscript and claim ledger must then narrow the scope.

17.8 Threats to benchmark validity

Compile-time microbenchmarks can be dominated by allocation, serialization, filesystem state, or debug builds. Execution benchmarks can be dominated by page cache, network placement, client decoding, or result size. Prepared statement tests can accidentally compare a cached native plan with an uncached CatDB path. PostgreSQL parameter skew can make one generic/custom plan policy misleading. SQLite explain output can change across versions.

Mitigations include randomized run order, separated process and connection lifetimes, fixed warmup rules, repeated distributions, recorded build modes, server/client clock separation where needed, exact result consumption, and published raw data. Benchmarks should report wall time and component measurements without subtracting inconvenient work.

18 Reproducibility protocol

18.1 Artifact bundle

Each experiment run creates an append-only bundle:

```
run/
  manifest.json
  environment.json
  artifacts/
    schema.json
    mappings.json
```

```
  query.json
  capability.json
  policy.json
fixtures/
  ddl.sql
  data-manifest.json
  seed.txt
compilation/
  logical-ir.json
  dispositions.json
  realization.json
  statement.sql
  parameters.json
execution/
  result.json
  lineage.json
  errors.json
  native-plan.json
  counters.json
  trace.json
comparison/
  oracle-result.json
  oracle-lineage.json
  verdict.json
metrics/
  observations.csv
  summary.json
checksums.txt
```

The bundle contains failures as well as successes. Raw source credentials, personal data, and secrets are excluded; synthetic fixtures are preferred for the first slice. The manifest points to exact repository and tool revisions and records whether the tree was dirty.

18.2 Independent oracle

The finite evaluator implements the paper denotation directly, without calling the SQL emitter or backend connector. Sharing scalar codecs may be practical, but any shared component is named because it reduces independence. Hand-computed tiny cases test the oracle. Property tests generate finite required-scalar inputs for scan, select, project, and join. The Haskell reference model may later serve as another oracle, but no such P2 oracle exists at the evidence cut.

18.3 Verification sequence

The required order is:

1. validate artifact schemas and hashes;
2. run positive and negative semantic fixtures;
3. compare the independent evaluator with hand cases;

4. run SQLite differential correctness and lineage;
5. run SQLite cache and hot-path traces;
6. run PostgreSQL portable-subset correctness and lineage;
7. run parameter and plan-policy experiments;
8. run fair native performance comparisons;
9. reproduce from a clean environment; and
10. submit the frozen manuscript and artifact digest to domain, database-systems, adversarial, code, reproducibility, and readability review.

No performance claim is accepted before correctness and lineage gates pass. No paper claim is labeled machine-checked merely because a Lean file compiles some P1 definition.

18.4 Statistical reporting

Latency reports include sample count, warmup and exclusion rules fixed in advance, median, selected quantiles, dispersion, and confidence interval or bootstrap procedure. Throughput reports identify concurrency, client count, backpressure, result consumption, and duration. A result table links each summary number to raw observations. Multiple machines or backend versions are reported separately before any aggregate.

19 Current implementation status

As of the 23 July 2026 evidence cut, this paper is a provisional formal and systems design. There is no accepted P2 vertical slice. Specifically:

- no versioned query-v1 serialized profile has been implemented;
- no independent finite query evaluator has been implemented;
- no Rust `catdb-query` or `catdb-planner` evidence is present;
- no Rust `catdb-storage-sqlite` or `catdb-storage-postgres` evidence is present;
- no compiler-generated SQL or typed parameter artifact is present;
- no SQLite or PostgreSQL native plan has been captured for a CatDB query;
- no normalized result or lineage differential has been run;
- no semantic/realization/backend cache transition suite has been run;
- no hot-path trace or performance benchmark has been run;
- no Haskell P2 query oracle exists; and
- no Lean theorem checks the P2 query or lowering laws.

The internally accepted P1 schema, path, and mapping slices supply bounded inputs to P2. They do not supply the missing query, realization, connector, differential, or measurement evidence. The SQL listings in this manuscript are specification examples, not captured compiler output.

19.1 Implementation roadmap

The minimum vertical slice is:

1. freeze the query-v1 schema and canonical fixtures;
2. implement the independent finite-bag evaluator;
3. add immutable query and capability artifacts to the Rust semantic IR;
4. implement versioned lineage modes and measure required versus permitted-minimal projection width before backend optimization;
5. implement typed expansion and normalization for Q_{co};
6. implement exact SQLite capability checking, SQL emission, typed binding, result decoding, and lineage;
7. pass `P2-E01`–`P2-E10` for SQLite;
8. implement the PostgreSQL portable-subset connector;
9. pass `P2-E04`, `P2-E11`, and the cross-backend subset;
10. add reproducible benchmark packaging; and
11. only then revise this paper with measured results.

Graph, document, RDF, vector, Arrow, Substrait, and polystore work begins only after the relational correctness contract is stable enough to expose what must be generalized.

20 Limitations and threats to validity

20.1 Narrow query language

Query-v1 excludes many operations required by real analytics: nulls, outer joins, aggregation, grouping, ordering, limits, windows, recursion, updates, and approximate numeric semantics. The finite-bag theorem becomes useful only as these features receive explicit semantics. Expanding the grammar will add proof and backend obligations; it should not be done by treating SQL syntax availability as equivalence.

20.2 Total functional path bias

The schema profile's arrows are total functions. Many real relationships are optional, many-to-many, temporal, uncertain, or contradictory. Encoding them as bridge entities may be

possible but changes query shape and user meaning. Property graphs and RDF can represent relationships that do not fit this bounded profile naturally.

20.3 Mapping-witness burden

An explicitly directional read witness prevents unsafe inversion, but it requires substantial authoring and validation. Automatic schema matching could propose assignments, yet suggestions do not establish meaning. The program has not measured whether witness reuse reduces maintenance enough to justify the additional semantic artifacts.

20.4 Backend-semantic gap

The abstract lowering theorem assumes exact primitive realizations. Real SQL engines have nulls, coercions, collations, transaction effects, errors, planner changes, and driver behavior. Differential tests can find mismatches but do not prove all executions correct. Formal refinement to a precisely modeled SQL subset would strengthen the result at substantial cost.

20.5 Native-plan observability

Explain and counters reveal only what the backend exposes. PostgreSQL `EXPLAIN ANALYZE` changes measurement conditions and executes the query; SQLite explain formats are unstable across versions [16, 24, 25]. Absence of an observed semantic callback is not by itself proof that every runtime path avoids one. Profiling, call counting, and code inspection must agree.

20.6 Performance generalization

SQLite and PostgreSQL results on synthetic Customer/Order data will not generalize automatically to remote federation, graph traversal, document unnesting, vector search, or distributed execution. Hardware, data skew, indexes, driver choices, cache state, and result size can reverse comparisons. The evaluation therefore scopes every result to a manifest and workload.

20.7 Provenance cost

Lineage improves auditability but can increase projection width, transfer, storage, and comparison cost. The first lineage form is intentionally bounded. Full provenance semantics and compression strategies belong to P13. P2 must compile the explicit lineage mode, reject any silent downgrade, and report required-lineage and policy-permitted minimal-lineage execution separately rather than hiding the cost. No zero-cost abstraction or negligible overhead is assumed; early fixtures and microbenchmarks record added columns, bytes, allocations, and latency before an optimization claim is allowed.

20.8 No update semantics

The paper addresses reads. An exact read mapping need not support safe write-back. Inversion, lenses, target constraints, authority, conflicts, and multi-source ambiguity belong to the bidirectional-mappings program. Nothing here authorizes writes through a canonical query.

20.9 No distributed-consistency result

The snapshot premise in Theorem 10.2 is local and abstract. Cross-source consistent snapshots, replica convergence, causal ordering, and CAP-style tradeoffs are not solved by semantic compilation. A federated result must state its consistency and freshness contract; P2 does not provide one.

21 Open questions

1. What smallest scalar profile has identical, useful behavior across the finite evaluator, Rust, SQLite, PostgreSQL, and later Haskell?
2. Which wider-precision or mixed-scale decimal profile can extend $\mathsf{Decimal64}(p, s)$ without weakening cross-backend exactness?
3. Which read-rewrite witnesses can be derived from lawful categorical mappings, and which require explicit view definitions?
4. What decidable fragment of path equations is useful for query normalization without introducing an unbounded equality procedure?
5. When does equation-based rewriting reduce work, and when does it make planner estimates or provenance harder to interpret?
6. How should semantic and physical statistics be versioned so that a realization cache is neither unsound nor needlessly cold?
7. Which SQLite binding exposes preparation flags, statement counters, cancellation, and streaming with the least hidden work?
8. Which PostgreSQL driver exposes typed codecs, cancellation, streaming, prepared-plan policy, and structured evidence sufficiently for the reproducibility bundle?
9. How should a residual evaluator enforce memory and backpressure limits while preserving bag and error semantics?
10. Can a normalized lineage algebra remain compact enough for ordinary interactive queries?
11. Is Arrow beneficial at the embedded SQLite boundary, or should it be reserved for process and network exchange?

12. Can Substrait represent the admitted logical plan without a custom extension, and what CatDB evidence necessarily remains outside it?
13. Which property-graph path fragment corresponds exactly to the total functional-arrow profile?
14. How should document missing values, arrays, and heterogeneous fields enter an extended type system?
15. What quality and provenance contract is sufficient for an ANN operator to participate safely as a candidate generator?
16. What benchmark demonstrates that versioned mappings reduce repair surface rather than merely relocating configuration?
17. Which paper-level lowering lemmas have enough value and stable scope to mechanize in Lean?

These are not rhetorical future-work items. Each can change the admissible profile, cache keys, evidence format, or central performance claim.

22 Conclusion

The physical-compilation claim has a narrow boundary. Typed, versioned mappings expand a canonical query into source meaning, and the logical IR fixes its records, finite bags, predicates, projections, and inner joins. A versioned capability manifest marks each operation as exact, residual, approximate, or unsupported. The resulting realization contains a parameterized statement and its evidence. SQLite or PostgreSQL still chooses the scans, indexes, joins, and row-execution details.

The three paper proofs cover the abstract design: well-witnessed path expansion preserves endpoints, exact compositional lowering preserves the admitted finite-bag denotation, and dependency-complete cache keys permit sound reuse under the stated assumptions. These are not compiler or backend results. The repository still lacks the query language, evaluator, planner, connectors, generated statements, differential results, native-plan captures, hot-path traces, benchmarks, Haskell oracle, and Lean query theorem needed to validate the systems claim.

The next credible milestone is one complete Customer/Order vertical slice. It needs immutable inputs, typed expansion, SQLite emission, independent result and lineage equivalence, explicit cache states, native evidence, and a trace showing that semantic work does not scale with the row count for a fixed query shape. PostgreSQL parity would then use the same portable contract. Graph, document, vector, Arrow, Substrait, and polystore claims remain preservation analysis until those gates pass and each later backend supplies its own evidence.

A Formal typing rules for the core fragment

Let Σ map relations and aliases to record types, and let Γ map parameters to scalar domains. Expression typing has the expected rules:

$$\frac{\ell : \tau \in \rho}{\Gamma; \rho \vdash \ell : \tau}, \qquad \frac{x : \tau \in \Gamma}{\Gamma; \rho \vdash : x : \tau}.$$

For a declared total comparison on τ:

$$\frac{\Gamma; \rho \vdash e_1 : \tau \qquad \Gamma; \rho \vdash e_2 : \tau}{\Gamma; \rho \vdash e_1 \geq e_2 : \mathsf{bool}}.$$

No coercion rule is present. A comparison between decimal and text is ill-typed even if a backend would coerce it.

Query typing includes:

$$\frac{\Sigma(R) = \rho}{\Sigma; \Gamma \vdash \mathsf{Scan}(R, \alpha) : \mathcal{B}_{\mathrm{fin}}(\alpha.\rho)}$$

and

$$\frac{\Sigma; \Gamma \vdash Q : \mathcal{B}_{\mathrm{fin}}(\rho) \qquad \Gamma; \rho \vdash \varphi : \mathsf{bool}}{\Sigma; \Gamma \vdash \mathsf{Select}_{\varphi}(Q) : \mathcal{B}_{\mathrm{fin}}(\rho)}.$$

If projection expressions e_i have types τ_i, then:

$$\frac{\Sigma; \Gamma \vdash Q : \mathcal{B}_{\mathrm{fin}}(\rho) \qquad \Gamma; \rho \vdash e_i : \tau_i \ (1 \leq i \leq n)}{\Sigma; \Gamma \vdash \mathsf{Project}_{\ell_i = e_i}(Q) : \mathcal{B}_{\mathrm{fin}}(\langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle)}.$$

For join:

$$\frac{\Sigma; \Gamma \vdash Q_1 : \mathcal{B}_{\mathrm{fin}}(\rho_1) \quad \Sigma; \Gamma \vdash Q_2 : \mathcal{B}_{\mathrm{fin}}(\rho_2) \quad \Gamma; \rho_1 \times \rho_2 \vdash \theta : \mathsf{bool}}{\Sigma; \Gamma \vdash \mathsf{Join}_{\theta}(Q_1, Q_2) : \mathcal{B}_{\mathrm{fin}}(\rho_1 \oplus \rho_2)}.$$

The labels of ρ_1 and ρ_2 must be disjoint after alias qualification.

B Running-query derivation sketch

Let

$$\rho_o = \langle o.id : \mathsf{uuid}, o.customer\_id : \mathsf{uuid}, o.total : \mathsf{decimal} \rangle,$$
$$\rho_c = \langle c.id : \mathsf{uuid}, c.name : \mathsf{string} \rangle.$$

The two scans have types $\mathcal{B}_{\mathrm{fin}}(\rho_o)$ and $\mathcal{B}_{\mathrm{fin}}(\rho_c)$. Both sides of $o.customer\_id = c.id$ have type uuid, so the join predicate is total Boolean under the required-field profile. The join has type $\mathcal{B}_{\mathrm{fin}}(\rho_o \oplus \rho_c)$. The parameter context contains $: minimumTotal : \mathsf{decimal}$, so $o.total \geq : minimumTotal$ is a typed total predicate. The four projected expressions have types

$$\mathsf{uuid}, \quad \mathsf{string}, \quad \mathsf{uuid}, \quad \mathsf{decimal}.$$

Consequently L_{co} has the output type:

$$\mathcal{B}_{\mathrm{fin}} \left(\left\langle \begin{array}{l} customerId : \mathsf{uuid}, \\ customerName : \mathsf{string}, \\ orderId : \mathsf{uuid}, \\ orderTotal : \mathsf{decimal} \end{array} \right\rangle \right).$$

The derivation becomes invalid if either join key is nullable, the UUID codecs disagree, the decimal comparison is not exact, or the source relationship does not have the totality required by the navigation witness. Such facts cannot be repaired by the typing tree alone.

C Review-time claim checklist

Table 6: Claims to inspect during paper review.

Claim	Present label	Required promotion evidence
Typed path expansion preserves endpoints	proved in this paper	optional Lean theorem and fixture cross-check
Abstract exact lowering preserves bag denotation	proved in this paper	machine-checking plus primitive-refinement work for stronger claim
Dependency-complete cache reuse is sound	proved in this paper	canonicalization/hash tests and mutation suite
SQLite realization is correct	engineering hypothesis	`P2-E03` value, error, and lineage differential
PostgreSQL portable subset is correct	engineering hypothesis	`P2-E04` cross-backend differential
Semantic compilation stays outside row time	engineering hypothesis	`P2-E10` trace and call-count evidence
Cached overhead is below target	engineering hypothesis	`P2-E12` fair distributions
Mappings improve integration maintenance	open in P2	P4 integration-complexity experiment
Graph/document lowering is exact for a useful fragment	open	formal profile and differential connector evidence
Vector execution is safe as candidate generation	open	quality, provenance, and reconciliation study

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, et al. Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[2] Apache Arrow Project. Columnar format. Official format specification, version 1.5. Accessed 2026-07-22.

[3] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over

heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, 2018.

[5] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[6] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: Proving query rewrites with univalent sql semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 510–524, 2017.

[7] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[8] Goetz Graefe. Volcano—an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[9] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[10] ISO/IEC JTC 1/SC 32. Information technology—database languages—gql. Technical Report ISO/IEC 39075:2024, International Organization for Standardization, 2024.

[11] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[12] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 149–159, 1986.

[13] Yu A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[14] PostgreSQL Global Development Group. Postgresql 18: Constraints. Official documentation. Accessed 2026-07-22.

[15] PostgreSQL Global Development Group. Postgresql 18: Data types. Official documentation. Accessed 2026-07-22.

[16] PostgreSQL Global Development Group. Postgresql 18: Explain. Official documentation. Accessed 2026-07-22.

[17] PostgreSQL Global Development Group. Postgresql 18: Prepare. Official documentation. Accessed 2026-07-22.

[18] PostgreSQL Global Development Group. Postgresql 18: Query planning. Official documentation. Accessed 2026-07-22.

[19] P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[20] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[21] SQLite Project. Compiling an sql statement. Official C interface documentation. Accessed 2026-07-22.

[22] SQLite Project. Create table. Official documentation. Accessed 2026-07-22.

[23] SQLite Project. Datatypes in sqlite. Official documentation. Accessed 2026-07-22.

[24] SQLite Project. Explain. Official documentation. Accessed 2026-07-22.

[25] SQLite Project. Explain query plan. Official documentation. Accessed 2026-07-22.

[26] SQLite Project. Json functions and operators. Official documentation. Accessed 2026-07-22.

[27] SQLite Project. Prepared statement status. Official C interface documentation. Accessed 2026-07-22.

[28] SQLite Project. Sqlite foreign key support. Official documentation. Accessed 2026-07-22.

[29] SQLite Project. Status parameters for prepared statements. Official C interface documentation. Accessed 2026-07-22.

[30] SQLite Project. Strict tables. Official documentation. Accessed 2026-07-22.

[31] Substrait Project. About substrait. Official project description. Accessed 2026-07-22.

[32] Substrait Project. Substrait extensions. Official extension specification. Accessed 2026-07-22.

[33] Substrait Project. Substrait specification. Official specification. Accessed 2026-07-22.

[34] Trino Software Foundation. Pushdown. Official Trino documentation. Accessed 2026-07-22.

[35] W3C RDB2RDF Working Group. R2rml: Rdb to rdf mapping language. W3c recommendation, World Wide Web Consortium, 2012.

[36] W3C RDF Working Group. Rdf 1.1 concepts and abstract syntax. W3c recommendation, World Wide Web Consortium, 2014.

[37] W3C SPARQL Working Group. Sparql 1.1 query language. W3c recommendation, World Wide Web Consortium, 2013.

[38] Jianguo Wang, Xiaomeng Yi, Ruoyu Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yutian Zou, Jiachun Long, Yufei Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Rui Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[39] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

Boundary	Candidate preservation	Typical residual or loss	Status
Relational targets	required scalar fields, total arrows as key relationships, projection, filter, inner equijoin, selected native constraints	arbitrary path laws, null/missing semantics, undeclared collation, cross-source identity	initial relational target
Property graph/GQL	entity and relationship types, selected path traversals, explicit edge properties	edge identity, path multiplicity, variable-length path semantics, graph-specific constraints	design analysis only
RDF/SPARQL	named resources, predicates, graph patterns, mapping-based access	open-world assumptions, blank nodes, term construction, entailment regime, dataset scope	design analysis only
Document	object fields, arrays with declared order or bag policy, selected nested navigation	missing versus null, heterogeneous shapes, array position, duplicate field behavior	design analysis only
Vector/ANN	vector-valued attributes, metric and index request, candidate provenance	approximation, recall, nondeterministic ties, training/index state, identity inference	approximate candidate only
Polystore	engine assignment, casts, data/query shipping, island boundaries	cast loss, snapshot mismatch, source autonomy, cross-engine cost error	future planning work
Arrow	typed record-batch exchange	semantic mappings, query intent, constraints, snapshots	interchange candidate only
Substrait	portable relational-plan structure and extensions	CatDB mapping provenance, backend extension parity, exact scalar profile	plan-interchange candidate only

Table 4: Preservation matrix. Each non-relational row is a research boundary, not implementation evidence.