SAGE: A Unified Algebra and Self-Adaptive Execution for AI Functions in SQL

Xiangqi Wang[1,2] **Nhan H. Pham**[2] **Oktie Hassanzadeh**[2]
Dharmashankar Subramanian[2] **Xiangliang Zhang**[1]
[1]University of Notre Dame | [2]IBM Research

SQL systems increasingly expose AI functions for tasks such as classification, extraction, filtering, ranking, retrieval, joining, and summarization. Despite their diverse APIs, these functions play only three relational roles: transforming individual rows, aggregating groups, or generating relationships between row pairs. We present SAGE (Self-Adaptive Generative Execution), a unified logical and physical framework that captures these roles with three typed primitives, `AI_SCALAR`, `AI_AGG`, and `AI_JOIN`, and composes them naturally with standard relational operators. All primitives share a confidence-gated execution interface while supporting physical strategies tailored to their relational shape. The main challenge is `AI_JOIN`, where SAGE analyzes the predicate, decomposes compound conditions when possible, and uses a recipe card together with a small label-free probe to select among complete execution strategies. Across a broad audit of public AI operators and evaluations spanning scalar, aggregate, and join workloads, this formulation covers common AI functionality while consistently improving execution quality and efficiency. SAGE achieves the strongest overall SemBench performance and, on a representative factorable join, reduces pairwise model calls by more than two orders of magnitude, yielding a 358× measured cost reduction.

Correspondence xwang76@nd.edu
Date August 2026

1 Introduction

AI functions bring unstructured data into SQL. A user can classify a review, extract a field from a report, retrieve passages relevant to a question, match records under a natural-language condition, or summarize all documents in a group without leaving a relational query Patel et al. [2024], Liu et al. [2024a], Shankar et al. [2024]. Research systems and production warehouses consequently expose a growing vocabulary of `AI_CLASSIFY`, `AI_EXTRACT`, `AI_FILTER`, `AI_SIMILARITY`, `AI_JOIN`, and `AI_AGG`-like functions Databricks [2025], Snowflake [2025], Google Cloud [2025].

This operator growth hides a simpler execution structure. Rather than organizing AI functions by user-facing names, the query plan can reason about where a model invocation appears in relational dataflow. Each invocation plays one of three roles: `AI_SCALAR` maps a row to a value, `AI_AGG` reduces a group to a value, and `AI_JOIN` decides which row pairs survive. Classification, extraction, rewriting, and filtering are scalar operations; summarization is aggregation; retrieval, similarity search, entity resolution, and natural-language joins are pairwise predicates followed by standard relational operators. More complex tasks are compositions of these primitives.

These roles also induce different physical optimization problems. Scalars mainly require per-row model routing, aggregates additionally require compaction and chunking, and joins must control a potentially quadratic pair space and determine when predicates can be factorized. Across all three, model cost dominates execution: using only a strong model is expensive, while using only a cheap model sacrifices quality Chen et al. [2023]. Fixed cascades are also insufficient because difficulty varies across inputs, model quality is not always monotone with size Ong et al. [2024], and each primitive exposes different optimization choices. Thus, efficient execution requires a shared adaptive interface together with role-specific physical strategies.

We therefore separate the problem into a logical layer and a physical layer. The logical layer gives every query-time AI function a typed formulation and compiles surface syntax to a plan over three primitives plus ordinary SQL. The physical layer preserves those semantics while adapting effort. All primitives share a small-to-large cascade gated by answer-bearing token confidence Duan et al. [2024]. They differ only at their front ends: `AI_SCALAR` exe-

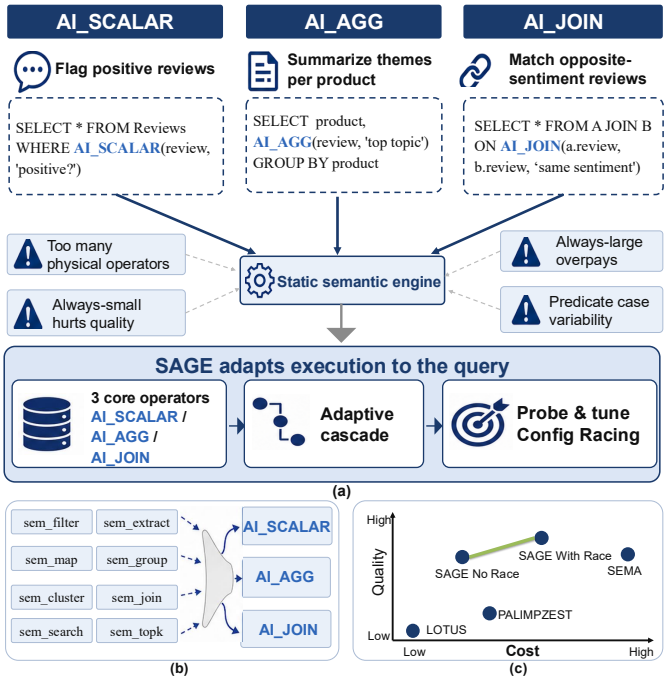


Figure 1: System overview. Surface AI functions compile to three logical primitives. Each primitive has a shared adaptive skeleton and a shape-specific physical front end; a per-query probe chooses the final execution configuration.

cutes rows directly, `AI_AGG` can compact redundant group members, and `AI_JOIN` uses a recipe-card probe to select membership, relational, or reasoning execution. A second, per-query probe evaluates complete configurations under the current predicate and freezes the best one satisfying cost and latency budgets. Figure 1 summarizes the design.

This organization changes the role of semantic join. The original join-only view made membership, relational, and reasoning predicates the top-level taxonomy. In the unified formulation they are physical subclasses of `AI_JOIN` only. The top level is instead the three-way division by relational cardinality: row preserving, group reducing, and pair generating. The same quality–cost objective, confidence signal, configuration interface, and test-time adaptation loop can therefore serve the full AI-function surface.

We realize this design in SAGE (Self-Adaptive Generative Execution) and separate two kinds of evidence. For logical breadth, an audit of 78 operators from eleven public systems maps 63 model-invoking operators directly to the three primitives; the remainder are multimodal conversions or non-semantic indexing and control constructs. Separately, 37 semantic-query intents require at most three primitives, and 94.6% require at most two. The unified evaluation reports SemBench Q1–Q10 Lao et al. [2025] and Multi-XScience Lu et al. [2020], thereby covering scalar, aggregate, join, and ranking workloads in one table. For overlapping joins we use the current AI-Join runs on SemBench Q5–Q7, FewRel Han et al. [2018], and BRIGHT Su et al. [2025]; three compound joins are reported separately. SAGE obtains the best SemBench average and the best or tied-best quality on every updated join except Q5, while the fixed no-racing variant gives the cheapest point on all three compound workloads. The join specialization reduces one exact all-pairs execution from 16,256 generative calls to 128, a measured 358× cost reduction.

Our contributions are:

1. A unified formulation of AI functions. We define three typed logical primitives by their relational cardinality and dependency signatures, give compilation rules for common AI-function APIs, and state the boundary of the formulation. This separates user-facing function names from the minimal model-invoking core.

2. Primitive-aware adaptive execution. We factor physical execution into a shared confidence-gated cascade and primitive-specific front ends. In particular, semantic-join routing and predicate decomposition become one specialization of the pair primitive, alongside scalar and aggregate plans.

3. Per-query configuration selection. We formulate execution as constrained configuration selection over an entire AI-function plan. A label-free probe races operator-specific candidates and freezes a quality-maximizing plan under cost and latency budgets.

4. Cross-primitive evaluation with a pair-primitive deep dive. We evaluate scalar, aggregate, join, and ranking workloads together on SemBench Q1–Q10 and Multi-XScience. For the overlapping join columns and three compound predicates, we use the latest AI-Join runs; these runs also support the mechanism ablations, recipe and proxy audits, slate analysis, convergence, scaling, and cross-domain evaluation of the pair primitive.

2 A Unified Algebra for AI Functions

We first define the logical layer independently of any model, prompt, or execution policy. Let a relation R contain tuples r, let I be a natural-language instruction, and let T be an SQL or structured output type. An AI function is a typed, model-evaluated expression whose relational role is determined by its input binding and cardinality effect. This produces three primitive signatures.

2.1 Three Logical Primitives

`AI_SCALAR`: row → value. The scalar primitive evaluates each already-bound tuple independently:

$$\mathsf{S}_{I,T} : r \in R \longmapsto v \in T. \tag{1}$$

Applied to a relation, it preserves cardinality, $N \to N$, by appending v as a virtual column. Classification, extraction, rewriting, scoring, question answering over row-local context, and Boolean filtering all instantiate Equation 1. A Boolean result can be consumed by `WHERE`; other types can appear in `SELECT`, `ORDER BY`, or a later primitive.

```
SELECT review_id,
       ai_classify(review_text,
         ARRAY['positive','negative','neutral']) AS
    sentiment
FROM reviews;
```

`AI_AGG`: group → value. The aggregate primitive evaluates a bag of tuples sharing a relational group key:

$$\mathsf{A}_{I,T} : \mathcal{B}(R_k) \longmapsto v_k \in T, \tag{2}$$

where $\mathcal{B}(R_k)$ is the bag of rows in group k. It reduces cardinality from N rows to G group outputs. Summaries, group-level judgments, semantic counts or estimates, consensus extraction, and structured theme lists are instances. Unlike

Primitive	Model scope	Cardinality	Main risk
AI_SCALAR	one bound row	$N \to N$	per-item error
AI_AGG	one group/bag	$N \to G$	lost evidence
AI_JOIN	one row pair	$NM \to K$	pair explosion

Table 1: The primitives are separated by their role in relational dataflow, not merely by prompt arity.

SUM or COUNT, A need not be associative or decomposable; that property belongs to its physical contract, not its logical signature.

```
SELECT movie_title,
       ai_agg(review_text, 'Summarize the audience
    sentiment.') AS summary
FROM reviews
GROUP BY movie_title;
```

AI_JOIN: pair → predicate. The join primitive evaluates a semantic condition over two independently ranged tuples:

$$\mathtt{J}_I : (a, b) \in A \times B \longmapsto y_{ab} \in \{0, 1\}. \tag{3}$$

Its relational result is

$$A \bowtie_I B = \{(a, b) \in A \times B : \mathtt{J}_I(a, b) = 1\}. \tag{4}$$

This primitive is cardinality-generating before selection: $|A| \times |B| \to K$. A physical implementation may produce a score and apply a threshold, but the logical contract is a pair predicate. Semantic join, entity matching, query–document relevance, similarity search, and top-k retrieval use the same pair judgment; ordinary SQL performs the final selection or ordering.

```
SELECT r.review_text, m.title
FROM reviews r JOIN movies m
  ON ai_join('Is this review about the movie?',
     r.review_text, m.title);
```

Why AI_JOIN is a Boolean judge over pairs. Any cross-table AI join can be viewed as a Boolean judgment over a candidate pair: given $r \in R$ and $s \in S$, the model decides whether they satisfy a semantic relation. This covers matching, relevance, similarity, membership, and natural-language relations. The key difference from AI_SCALAR is that AI_JOIN operates over an $N \times M$ pair space and determines which pairs survive. Exposing this pair space lets the optimizer enumerate, block, retrieve, or factor candidates before model invocation. Thus, AI_JOIN is essentially a semantic IF over row pairs, with the relational structure needed for join optimization.

2.2 Scope, Completeness, and Coverage

Our claim covers *single-pass, query-time semantic functions*, where each model call consumes relational data and returns a value or decision. This follows classical relational query theory, where complex queries are built by composing a small set of operators, including tuple and join functions Codd [1972] and group aggregation Klug [1982]. Under this view, every model call operates on one of three inputs: a row, a group, or a candidate pair, corresponding to AI_SCALAR, AI_AGG, and AI_JOIN. More complex tasks are compositions of these primitives with ordinary SQL. Recursive agents, training, multimodal conversion, indexing, and side-effecting actions are outside our scope.

System	#Ops	Scalar	Join	Agg	MM	NS
Databricks AI Functions Databricks [2025]	14	10	2	0	2	0
Snowflake Cortex AISQL Snowflake [2025]	12	5	2	2	2	1
LOTUS Patel et al. [2024]	11	6	3	1	0	1
BigQuery AISQL Google Cloud [2025]	11	8	1	0	1	1
DocETL Shankar et al. [2024]	9	3	2	1	0	3
FlockMTL Dorbani et al. [2025]	8	4	0	2	0	2
Palimpzest Liu et al. [2024a]	6	2	2	1	0	1
ThalamusDB Jo and Trummer [2024]	3	1	1	1	0	0
SUQL Liu et al. [2024b]	2	0	0	2	0	0
UQE Dai et al. [2024]	1	0	0	1	0	0
Generic agentic loop Madaan et al. [2023]	1	0	0	0	0	1
Union	**78**	**39**	**13**	**11**	**5**	**10**

Table 2: Audited compilation of public operator surfaces. MM denotes multimodal conversion and NS denotes non-semantic infrastructure outside the model-invoking core.

We further test this formulation against existing systems and workloads. Table 2 compares 78 operators from eleven public semantic-query systems. Among them, 63 semantic functions map directly to our three primitives, while the rest are conversion, indexing, ordinary SQL, or loop operations outside the scope. Appendix J.1 provides the full mapping. We also decompose 37 end-to-end semantic tasks into primitive plans. Figure 2 shows that almost all require no more than two AI primitives, and none requires more than three. Together, these results show that the three primitives cover both the existing AI-function surface and their common compositions in practical queries.

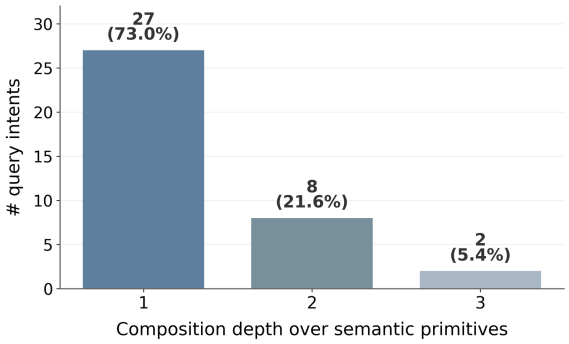


Figure 2: Coverage of 37 semantic-query intents by primitive composition depth. Most require one or two primitives; all require at most three.

2.3 Plan-Level Optimization Objective

Let a compiled query contain AI nodes $F_q = (f_1, \ldots, f_h)$, where $f_i \in \{\mathsf{S}, \mathsf{A}, \mathsf{J}\}$, and let $a = (a_1, \ldots, a_h)$ select one physical configuration for each node. We evaluate the complete plan by task quality $Q(q, a)$, monetary cost $C(q, a)$, and end-to-end latency $L(q, a)$. Given budgets B_C and B_L, the optimizer seeks

$$\begin{aligned} a_q^\star &= \arg\max_{a \in \mathcal{A}(F_q)} Q(q, a) \\ \text{s.t.} \quad & C(q, a) \le B_C, \qquad L(q, a) \le B_L. \end{aligned} \tag{5}$$

This plan-level form matters. A locally accurate scalar extraction may materially reduce join cost downstream; an early lossy join or aggregate may irreversibly remove evidence. SAGE therefore materializes reusable scalar values, pushes lossless and selective predicates early, and avoids partial execution of holistic aggregates. Appendix J gives the composition and MapReduce conditions used by the optimizer.

3 Primitive-Aware Physical Execution

The logical layer defines what each AI function means, while the physical layer decides how to execute it efficiently. SAGE shares confidence, escalation, and configuration mechanisms across primitives, while preserving the distinct optimization needs of rows, groups, and pair spaces.

3.1 A Shared Adaptive Skeleton

For a logical invocation $f_I(x)$, let M_f be an optional cheap front end, M_s the model assigned the first generative attempt, and M_ℓ the model assigned uncertain cases. The subscripts indicate routing roles, not model size: empirical quality need not increase monotonically with parameter count. Execution has three decisions:

1. M_f may settle or prune an input when its margin exceeds a front-end cutoff τ_f;
2. otherwise M_s produces an answer $\hat{y}_s$ and confidence $s(x, \hat{y}_s)$;
3. SAGE returns $\hat{y}_s$ when $s(x, \hat{y}_s) \geq \tau_e$ and asks M_ℓ to re-evaluate the input otherwise.

For a chosen physical plan, let n_f be the number of front-end evaluations, n_s the number of first-model evaluations, and n_ℓ the number of escalations. The model-call component of cost is

$$C_{\text{model}} = n_f\, c_f + n_s\, c_s + n_\ell\, c_\ell, \tag{6}$$

with zero terms for absent stages. The primitive-specific physical plan determines n_f and n_s—for example, a factorable join can replace NM pair evaluations with $N + M$ row evaluations—while the confidence gate controls n_ℓ.

Answer-bearing confidence. For an output $y = (y_1, \ldots, y_T)$, a standard confidence score averages token log-probabilities:

$$C_{\text{LP}}(y) = \frac{1}{T} \sum_{t=1}^{T} \log p(y_t \mid y_{<t}, x). \tag{7}$$

This score can be dominated by confident but uninformative tokens. SAGE instead uses TokenSAR Duan et al. [2024], which gives more weight to tokens that matter to the answer:

$$C_{\text{TokenSAR}}(y) = \sum_{t=1}^{T} \tilde{R}_t \log p(y_t \mid y_{<t}, x), \qquad \tilde{R}_t = \frac{R_t}{\sum_{j=1}^{T} R_j}, \tag{8}$$

where R_t measures the relevance of token y_t to the output meaning. Label-valued functions use the answer-token confidence directly, while free-form functions use the relevance-weighted score. The query probe calibrates the escalation threshold τ_e. In our AI-Join runs, this signal separates correct and incorrect answers more reliably than mean log-probability and improves routing over random escalation under the same large-model budget (Figures 14 and 15). More broadly, escalation is conditional rather than global: SAGE chooses the model that performs best on uncertain cases instead of assuming that the largest model is always strongest (Table 11).

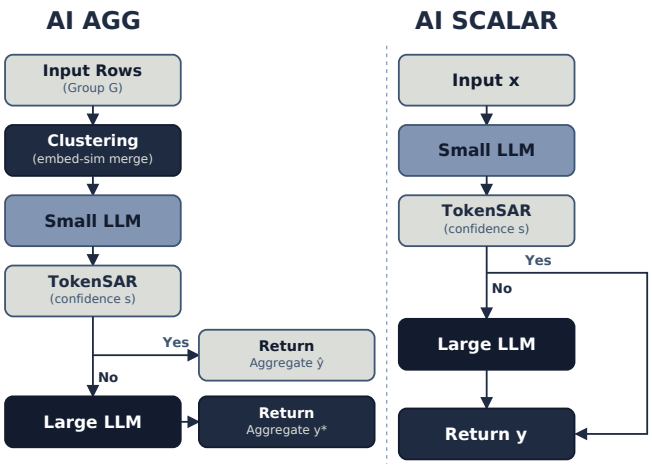


Figure 3: Shared cascade for `AI_SCALAR` and `AI_AGG`. Scalar calls enter per row; aggregate calls enter per group after optional compaction. Low-confidence outputs alone reach the escalation model.

3.2 AI_SCALAR: Independent Row Execution

For `AI_SCALAR`, the logical items are existing row bindings, so calls factorize exactly. SAGE batches or parallelizes rows without changing semantics, runs the assigned first model, and escalates only low-confidence outputs. A cheap labeler can serve as M_f for closed-set classification; for free-form extraction or rewriting the front end is absent and all rows begin at M_s. The configuration is

$$a_{\text{S}} = (M_f, M_s, M_\ell, \tau_f, \tau_e, \text{reasoning mode}, T, \text{batch size}).$$

The decisive property is that one row's answer does not depend on another row. Thus caching, vectorized inference, and pass-rate ordering of consecutive Boolean scalars are exact. If several downstream nodes reuse a generated field, SAGE materializes it once instead of regenerating it inside every pair or group.

3.3 AI_AGG: Group Reduction

For `AI_AGG`, one logical item is a complete SQL group. Groups often contain near-duplicate evidence, so the optional front end embeds group members, clusters entries above cosine threshold τ_c, and represents each cluster once with a weight or exemplar. The compacted group enters the shared cascade. The configuration adds the compaction policy:

$$a_{\text{A}} = (M_s, M_\ell, \tau_c, \tau_e, \text{chunking}, \text{reasoning mode}, T).$$

Compaction and chunking have different correctness contracts. Near-duplicate compaction is an approximation whose effect is measured on the probe. Chunking can be exact only when the aggregate is modular or algebraic: there must exist a partial state h, merge operator $\oplus$, and finalizer g such that

$$h(G_1 \uplus G_2) = h(G_1) \oplus h(G_2), \qquad \text{A}(G) = g(h(G)).$$

Counts and sums over already materialized model-extracted values, as well as mergeable vote summaries, may satisfy this contract. Holistic median-like judgments and summaries depending on cross-document evidence generally do not. SAGE therefore keeps the full group for

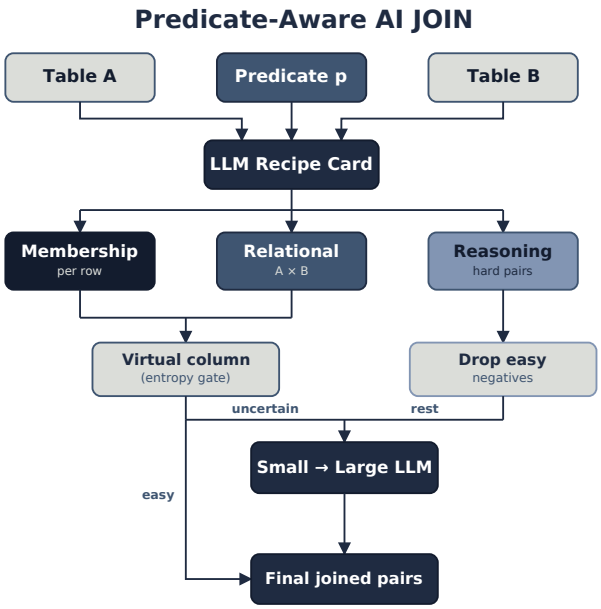


Figure 4: Predicate-aware `AI_JOIN`. A cached recipe card selects a membership, relational, or reasoning route; only the uncertain residual enters the shared cascade.

holistic aggregates rather than silently treating prompt chunking as an exact optimization. The measured decomposability study appears in Appendix J.

3.4 `AI_JOIN`: Predicate-Aware Pair Execution

`AI_JOIN` is the only primitive whose candidate domain may be quadratic. Its physical plan must be chosen before the cascade because the predicate determines whether $|A||B|$ pair judgments are necessary. The three join classes from the join-specific system are retained here as *physical routes inside* `AI_JOIN`, not as top-level AI-function primitives.

Virtual-column test. An LLM compiler reads the predicate and a small sample of candidate pairs and emits a cached recipe card. The card asks whether the predicate can be written as

$$p(a, b) \equiv \phi(f(a), g(b)), \tag{9}$$

where f and g are stable row-local virtual columns and ϕ is a deterministic comparison. If not, it asks whether a cheap pair model can score a closed relation label. These two axes select one of three routes:

$$\begin{aligned} \text{route}(p) &= \text{membership} && \text{if } p \equiv \phi(f(a), g(b)), \\ &= \text{relational} && \text{if a closed pair label exists,} \\ &= \text{reasoning} && \text{otherwise.} \end{aligned} \tag{10}$$

The card also records the featurizer, labels, negative check, and candidate configuration space. Figure 5 shows the compilation artifact; card validity and sample sensitivity are evaluated in Appendices A and A.1.

Membership route: materialize and hash-join. For "same sentiment," f and g independently assign labels in a closed set and ϕ tests equality. SAGE runs GLiClass Stepanov et al. [2025] or the scalar cascade once per row, materializes the two virtual columns, and applies

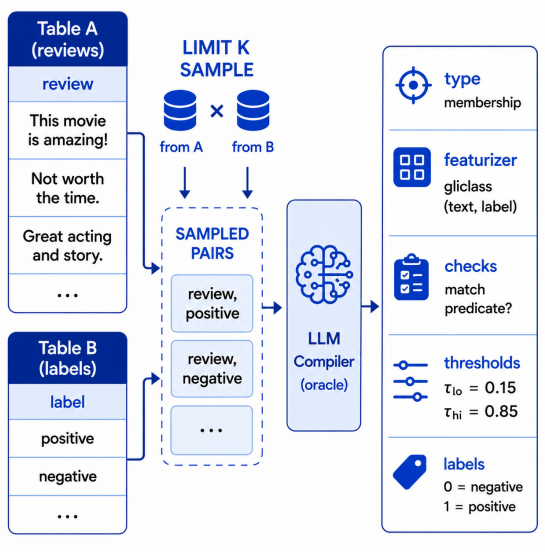


Figure 5: Recipe-card compilation. A small `LIMIT`-style sample and the predicate produce a cached, executable join route rather than a natural-language explanation alone.

an ordinary hash join. Predicate work drops from $|A||B|$ pair evaluations to $|A| + |B|$ row evaluations. The transformation is exact if Equation 9 and the predicted row values are correct; model error remains visible as ordinary virtual-column error.

Relational route: filter the pair grid. For "person a is the spouse of person b," no useful label belongs to either row independently. The predicate is a property of the pair, but a lightweight relation model such as GLiREL Boylan et al. [2025] can reject easy negatives. The pair grid remains logically present; only pairs above the front-end cutoff enter the generative cascade.

Reasoning route: conservative negative pruning. For "document a provides enough evidence to answer question b," no stable closed label captures the decision. Embedding similarity or a membership rewrite can remove valid matches. The recipe card supplies a self-descriptive negative condition; the front end commits only high-margin negatives, and all ambiguous pairs receive generative reasoning. This route spends more per survivor but avoids an unsafe factorization.

Compound predicates. A join condition may be $p = p_1 \wedge \cdots \wedge p_m$ with atoms on different routes. The compiler constructs one physical plan per atom and applies predicate pushdown. On a shared probe, it estimates pass fraction $\hat{\sigma}_i$ and amortized per-candidate cost $\hat{c}_i$, then orders atoms by

$$\frac{\hat{c}_i}{1 - \hat{\sigma}_i}, \tag{11}$$

the estimated cost per eliminated candidate. A cheap membership atom can therefore prune the pair space before a reasoning atom executes. This is where the original AI-Join decomposition fits the unified plan: it optimizes one `AI_JOIN` node and can itself consume `AI_SCALAR` virtual columns.

Outer joins and dangling rows. The model-invoking core still decides pair membership. `LEFT`, `RIGHT`, and `FULL` semantics are implemented by ordinary relational completion after the selected route, provided the engine tracks whether each preserved row received a qualifying match. Appendix G verifies dangling-row recall under selective predicates.

4 Test-Time Configuration Selection

4.1 Configuration Spaces and Slates

For node f_i, let $\mathcal{A}_{f_i}$ contain only configurations that preserve its logical contract. A scalar candidate bundles model roles, confidence threshold, reasoning mode, temperature, and batching. An aggregate candidate adds compaction and legally available partial-reduction choices. A join candidate adds recipe route, blocking or factorization parameters, and front-end cutoff. A query with AI nodes $F_q = (f_1, \dots, f_h)$ has the product space

$$\mathcal{A}(F_q) = \mathcal{A}_{f_1} \times \cdots \times \mathcal{A}_{f_h},$$

which is too large to execute exhaustively.

Each primitive therefore maintains a lightweight proposer. Given an embedded instruction, logical type, input statistics, and previous probe outcomes, the proposer emits a small candidate set containing (i) a cheap anchor, (ii) a quality anchor, and (iii) history-guided exploratory settings. For a multi-node query, SAGE assembles compatible node candidates into a slate $S_q \subset \mathcal{A}(F_q)$ of at most K whole-plan configurations. Cold start uses the anchors and space-filling samples; later queries reuse primitive-specific feedback. This proposal stage reduces the number of plans tested but does not decide the winner. Let $\mathcal{A}_q^{\text{feas}} = \{a \in \mathcal{A}(F_q) : C(q,a) \le B_C,\ L(q,a) \le B_L\}$ and $Q_q^\star = \max_{a \in \mathcal{A}_q^{\text{feas}}} Q(q,a)$ denote the budget-feasible oracle quality for query q. The proposer seeks

$$S_q = \arg\max_{\substack{S \subseteq \mathcal{A}(F_q) \\ |S| \le K}} \Pr\left[\max_{a \in S \cap \mathcal{A}_q^{\text{feas}}} Q(q,a) \ge (1-\varepsilon) Q_q^\star\right]. \tag{12}$$

4.2 Probe-and-Race Execution

SAGE draws a small stratified probe P_q from the current data and evaluates every candidate on the same sampled rows, groups, or candidate pairs. For composed queries, the sampled subplan is executed far enough to capture downstream effects. On each probe block, SAGE records actual token usage, cost, and latency, uses a fixed judge model to obtain the label-free quality estimate $\widehat{Q}_q(a)$, and extrapolates resource usage to the full query. An iRace-style procedure López-Ibáñez et al. [2016] then progressively eliminates statistically inferior or budget-infeasible candidates. Reusing the same probe across candidates ensures that differences reflect configuration rather than sampling variation. The final choice is

$$\hat{a}_q = \arg\max_{a \in A_q} \widehat{Q}_q(a) \quad \text{s.t.} \quad \widehat{C}_q(a) \le B_C, \quad \widehat{L}_q(a) \le B_L, \tag{13}$$

where A_q is the surviving slate. No benchmark gold labels are used at deployment, and the resulting probe outcomes update the proposer for future queries. Full pseudocode appears in Algorithm 1 (Appendix E).

Further validation of configuration racing. We further validate configuration racing on `AI_JOIN`. Figure 7 uses SemBench Q5 as a representative case: quality varies substantially across (τ_f, τ_e), showing that fixed thresholds can miss the best configuration. Additional results confirm that the selected configuration stabilizes with a small probe, the proposed slate achieves high near-oracle coverage, and the label-free proxy closely tracks a gold-based oracle.

5 Evaluation

We evaluate the complete three-primitive surface and then study the `AI_JOIN` specialization in depth. Table 3 combines SemBench Q1–Q10 and Multi-XScience; overlapping joins use the latest repeated AI-Join runs, as do the compound workloads and mechanism studies. We ask whether SAGE improves cross-primitive quality and efficiency (RQ1), executes confounded predicates near the cost of their selective atom (RQ2), requires each mechanism (RQ3), and obtains reliable label-free estimates (RQ4).

Setup. SemBench Lao et al. [2025] spans classification (Q1–Q2), aggregation (Q3, Q4, Q8), joins (Q5–Q7), and ranking (Q9–Q10); Multi-XScience (MXS) Lu et al. [2020] adds aggregation. Membership uses SemBench Q5–Q7 at scale factor 1000; relational and reasoning joins use the *spouse* relation of FewRel Han et al. [2018] and the TheoremQA split of BRIGHT Su et al. [2025]. Amazon–Google ER Köpcke et al. [2010], x-stance Vamvas and Sennrich [2020], and SciFact Wadden et al. [2020] provide compound predicates whose atoms have different classes. Within a workload, all systems share inputs and pricing; we report normalized quality, USD cost, and wall-clock latency. Full-suite traces use `granite-3.3-8b-instruct` IBM Granite Team [2025] and `gpt-oss-120b` Agarwal et al. [2025]. Current join runs pair `Granite-3.0-8B` Granite Team [2024] with `gpt-oss-120b`, use GLiClass Stepanov et al. [2025] and GLiREL Boylan et al. [2025], and compare with LOTUS Patel et al. [2024], Palimpzest Liu et al. [2024a], SEMA Qi et al. [2026], thDB Jo and Trummer [2024], and FDJ Zeighami et al. [2025] under the AI-Join protocol; each external system otherwise uses its native configuration. SAGE (No Racing) freezes one configuration; Appendix I gives full provenance and settings.

Cross-primitive and per-query gains (RQ1). Table 3 places all ten SemBench queries and MXS in one comparison. SAGE leads the SemBench average (0.908), ties on MXS, and has the best or tied-best updated-join quality except Q5; racing is most visible on Q6 and Q7, while SAGE (No Racing) retains a strong fixed quality–cost tradeoff. Gains are largest on Q2, Q7, Q9, FewRel, and BRIGHT. Under shift, SAGE recovers 80.8% of the target-trained ceiling, versus 73.5% for LOTUS and 60.1% for FDJ (Figure 9). Join F1 covers all N pairs; predictions for the n probe pairs are reused. End-to-end cost and latency include recipe generation, probes, judge calls, and execution on the remaining $N - n$ pairs.

Performance at Confounded Predicate (RQ2). Table 4 reports the confounded workloads separately.

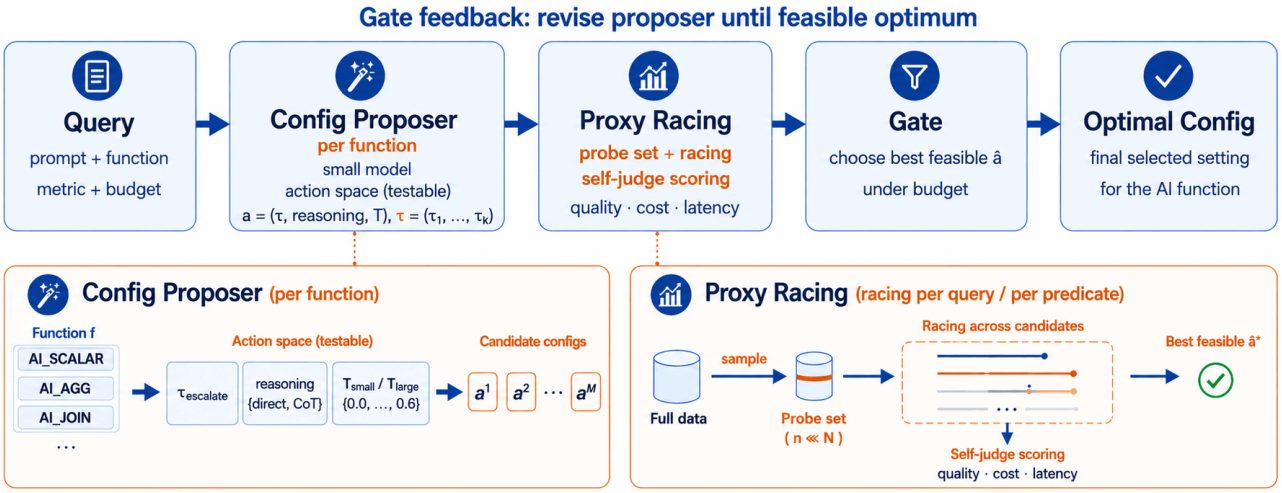


Figure 6: Probe-and-race execution. Primitive-specific proposers generate a small slate of semantically valid whole-plan configurations. A shared probe measures quality proxy, cost, and latency; racing eliminates inferior or infeasible candidates and freezes one plan for full execution.

	SemBench											Supplemental		
Method	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Summary	FewRel	BRIGHT	MXS
Task-normalized quality ↑														
Lotus	**1.00**	0.73	0.74	0.97	0.720	0.710	0.370	0.97	0.78	0.79	0.778	0.880	0.691	0.25*
Palimpzest	**1.00**	0.60	0.57	0.94	**0.944**	0.600	0.700	0.92	0.82	0.71	0.780	0.887	0.707	0.26
Sema	0.90	0.75	0.93	**0.99**	0.760	0.760	0.760	**0.99**	0.73	0.72	0.829	0.857	0.428	—
thDB	**1.00**	0.60	0.79	0.98	0.920	0.660	0.720	0.96	—	—	0.829†	0.880	0.563	—
FDJ	—	—	—	—	0.748	0.771	0.755	—	—	—	—	0.826	0.507	—
SAGE (No Racing)	0.98	0.70	**0.95**	0.95	0.850	0.770	0.750	0.98	0.70	0.80	0.843	0.868	0.774	**0.27**
SAGE	0.98	**0.92**	**0.95**	0.95	0.850	**0.850**	**0.850**	0.98	**0.91**	**0.84**	**0.908**	**0.902**	**0.805**	**0.27**
Cost (USD) ↓														
Lotus	0.210	0.012	0.012	0.013	6.100	5.650	5.640	0.012	0.030	0.210	17.890	0.013	0.011	0.0050*
Palimpzest	0.004	0.010	0.015	0.015	0.015	0.021	6.990	0.015	0.055	0.408	7.549	0.016	0.014	0.0006
Sema	0.130	0.007	0.008	0.007	2.950	2.940	2.940	0.007	0.030	0.200	9.220	0.011	0.010	—
thDB	**0.002**	0.006	0.010	0.010	0.006	0.006	0.236	0.010	—	—	0.284	0.011	0.011	—
FDJ	—	—	—	—	0.437	0.392	0.481	—	—	—	—	0.011	0.010	—
SAGE (No Racing)	0.0196	**0.0030**	**0.0068**	**0.0036**	**0.001**	**0.001**	**0.003**	**0.0040**	**0.0120**	0.1090	0.163	**0.001**	**0.005**	0.0004
SAGE	0.0196	0.0101	**0.0068**	**0.0036**	**0.001**	**0.001**	**0.003**	**0.0040**	0.0160	**0.0942**	**0.159**	**0.001**	0.009	**0.00008**
Latency (s) ↓														
Lotus	450.0	63.0	22.0	15.0	11356	12916	9511	15.0	32.0	748.0	35128	30.0	67.1	150*
Palimpzest	5.0	8.8	**13.5**	12.8	15.8	**22.6**	12960	13.1	48.4	427.9	13528	26.4	19.2	**4.3**
Sema	258.0	24.0	23.0	92.0	3895	3565	3711	20.0	51.0	417.0	12056	101.6	46.4	—
thDB	**2.7**	6.8	15.3	15.2	73.2	66.7	1970.6	12.6	—	—	2163	65.2	**16.1**	—
FDJ	—	—	—	—	1248	1187	1396	—	—	—	—	24.9	26.7	—
SAGE (No Racing)	51.5	**3.4**	36.7	**6.0**	**2.0**	25.0	**109.0**	**5.1**	18.2	**157.4**	**414.3**	**16.7**	18.8	12.7
SAGE	51.5	5.2	36.7	**6.0**	2.8	27.1	181.3	**5.1**	**6.0**	167.0	488.7	32.6	27.2	4.7

Table 3: Unified results on SemBench Q1–Q10 and supplemental workloads. Quality uses precision@5 (Q1–Q2), clipped 1 − relative error (Q3, Q4, Q8), F1 (Q5–Q7, FewRel, BRIGHT), a [0, 1] score transformed from Spearman correlation (Q9–Q10), and ROUGE-1 (MXS). The SemBench Summary is the arithmetic mean for quality and the sum for cost and latency, computed before displayed rounding. MXS cost and latency are per group. Q5–Q7, FewRel, and BRIGHT use the latest AI-Join repeated-run means; the other columns use the full-suite measurements. Dashes mark unsupported workloads; † averages only supported queries; * denotes an estimated infeasible LOTUS MXS run.

Selective-first execution gives SAGE the best mean F1 (0.846), while SAGE (No Racing) is cheapest on all three ($0.002–$0.007 versus $0.009–$0.031): the first atom prunes the pair space. Removing an atom or decomposition degrades precision or nearly doubles pairwise calls (Table 9, Appendix A).

Scalability Table 5 compares SAGE with an exact per-pair `AI_JOIN` plan on a representative join, both run to completion. It replaces 16,256 pairwise calls with 128 calls (127× fewer), cutting wall-clock by 89× and cost by 358×. In the Q6/Q7 sweeps (Figure 8), F1 stays stable while cost and latency fall by one to three orders of magnitude versus always-large execution.

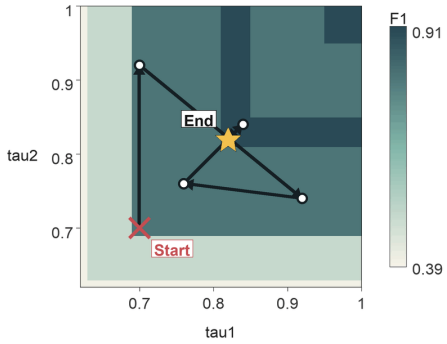


Figure 7: Configuration surface on SemBench Q5. Color shows measured F1 over the front-end and escalation thresholds (τ_f, τ_e), illustrating why SAGE selects configurations per query rather than using fixed thresholds.

Method	Amazon–Google	x-stance	SciFact
	F1 ↑		
Lotus	0.780±0.016	0.712±0.048	0.845±0.038
Palimpzest	**0.892±0.006**	0.687±0.011	0.861±0.015
Sema	0.812±0.031	0.726±0.022	0.864±0.000
thDB	0.838±0.007	0.690±0.014	0.864±0.000
FDJ	0.838±0.009	0.713±0.012	0.857±0.006
SAGE (No Racing)	0.810±0.000	0.719±0.004	0.861±0.003
SAGE	**0.892±0.007**	**0.755±0.013**	**0.891±0.005**
	Cost (USD) ↓		
Lotus	0.014±0.000	0.026±0.000	0.019±0.000
Palimpzest	0.015±0.000	0.031±0.000	0.021±0.000
Sema	0.014±0.000	0.029±0.003	0.012±0.001
thDB	0.011±0.000	0.023±0.000	0.015±0.000
FDJ	0.010±0.000	0.017±0.000	0.009±0.000
SAGE (No Racing)	**0.002±0.000**	**0.004±0.000**	**0.007±0.000**
SAGE	0.011±0.000	0.023±0.000	0.016±0.000
	Latency (s) ↓		
Lotus	123.5±145.5	131.4±101.4	183.2±141.0
Palimpzest	225.5±342.2	59.3±6.0	68.0±64.7
Sema	44.0±9.9	134.2±34.5	41.8±8.2
thDB	33.1±12.0	88.9±61.8	53.5±40.0
FDJ	30.6±2.8	55.8±4.3	**20.4±1.7**
SAGE (No Racing)	**17.2±2.0**	**22.6±1.5**	28.0±0.6
SAGE	27.6±5.6	34.5±4.9	44.5±6.4

Table 4: Confounded-predicate results, separated from the unified primitive table. Each workload conjoins two atoms of different classes; values are mean ± s.e.m. over the current AI-Join runs.

	LLM calls	Wall-clock	Cost
SAGE	128	53.4 s	$0.00166
Exact all-pairs plan	16,256	4,770 s	$0.595
Reduction	127×	89×	358×

Table 5: Predicate-aware `AI_JOIN` against the exact all-pairs plan on a representative join. Both plans were executed in full and serially; all three columns are measured, not projected.

Ablation across mechanisms (RQ3). Table 6 ablates one join per class; full SAGE gives the strongest overall quality–cost tradeoff. Mean log probability hurts membership and reasoning despite retaining escalation (F1 0.850 → 0.634 and 0.805 → 0.746). Removing routing is worse: the membership join becomes a $0.563 pairwise scan and FewRel falls to 0.039. The fixed no-racing variant also loses quality on Q2, Q6–Q7, Q9–Q10, FewRel, BRIGHT, and all three compound joins, while tying on the remaining Table 3 workloads. Neither model tier alone is both cheap and reliable; fixed-query interventions confirm the routing failure (Table 8, Appendix A).

The proposer fills the slate (RQ3). Racing can only choose from the slate, so a slate holding no good configuration cannot be repaired downstream. We replay 40 queries from each of four families after pre-executing all 96 configurations, so the oracle is known exactly. A slate hits when it contains a configuration within ε of the oracle (Eq. 12; protocol in Appendix F). At K=3, the learned proposer hits a near-oracle configuration on 60.8% of queries versus 3.4% for a uniform slate, outperforming ϵ-greedy, UCB Auer et al. [2002], and last-winner baselines (Figure 10). A cold-reset history falls to chance while a warm history sustains the hit rate; the proposer also reaches a near-oracle incumbent at low probe cost and remains reliable when most configurations are poor (Figure 17).

Convergence Pace The race stabilizes within about fifty probed samples, lifting deployed F1 from 0.685 to 0.746 (Figure 11); its overhead is bounded and front-loaded.

Validation of Estimators (RQ4). Proxy selection is compared with a gold oracle under the same candidates and budgets (Figure 12): proxy-selected configurations average 0.843 versus 0.892 for gold selection, with the largest gap on BRIGHT. Structural and sample-shift checks separately audit the recipe-card compiler (Table 7, Appendix A). These diagnostics quantify, rather than eliminate, label-free routing and selection error.

Cross-domain recovery. We train on one family and evaluate each held-out family without target labels, allowing only SAGE's unlabeled probe. Across ten stratified splits and twelve directions, recovery—target F1 divided by the best target-trained F1 among SAGE, LOTUS, and FDJ—is 80.8% for SAGE, 73.5% for LOTUS, and 60.1% for FDJ. Thus weak in-domain performance cannot inflate the score; SAGE leads or ties on eight directions (Figure 9).

6 Related Work

AI functions and relational structure. LOTUS, Palimpzest, ZenDB, Sema, ThalamusDB, DocETL, SUQL, and UQE expose model-backed mapping, filtering, extraction, retrieval, joining, and aggregation Patel et al. [2024], Liu et al. [2024a], Lin et al. [2024], Qi et al. [2026], Jo and Trummer [2024], Shankar et al. [2024], Dai et al. [2024]; production warehouses provide similarly broad AI-SQL suites Databricks [2025], Snowflake [2025], Google Cloud [2025]. Building on the logical–physical separation of relational optimization Codd [1970], Selinger et al. [1979], SAGE asks which properties must survive beneath these surface APIs. Its three signatures expose whether an invocation preserves rows, reduces a group, or tests a pair, thereby defining the semantic unit that each physical rewrite must preserve.

Structure-aware and adaptive execution. Semantic-join systems reduce the naive $|A||B|$ search through retrieval, batching, approximation, decomposition, or per-table factorization Patel et al. [2024], Jo and Trummer

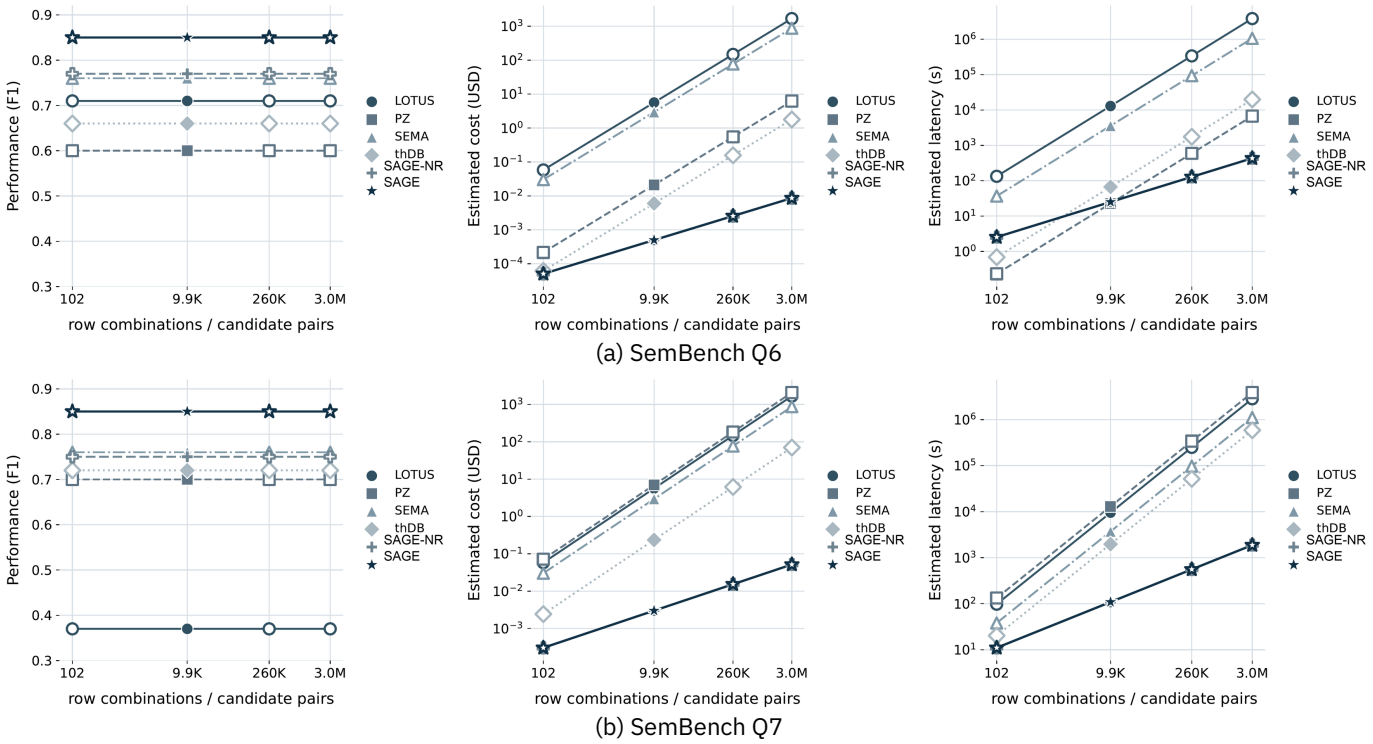


Figure 8: Measured scaling from 10^2 to 3×10^6 candidate pairs on SemBench Q6 and Q7. Left: F1; center: cost; right: latency. Each operating point is executed and scored under the AI-Join protocol.

	Membership (SemBench)		**Relational** (FewRel)		**Reasoning** (BRIGHT)	
Variant	F1	Cost ($)	F1	Cost ($)	F1	Cost ($)
Full SAGE	**0.850**	**0.003**	**0.902**	**0.001**	0.805	0.009
Mean-log-prob. gate	0.634	0.0079	0.881	0.0029	0.746	0.0138
w/o cheap front-end	0.755	0.0043	0.851	0.0130	**0.828**	0.0297
w/o predicate-aware routing	0.720	0.563	0.039	0.0191	0.630	0.0538
Large-only LLM	0.720	0.0155	0.851	0.0130	0.776	0.0225
Small-only LLM	0.034	0.0066	0.862	0.0022	0.646	**0.0073**

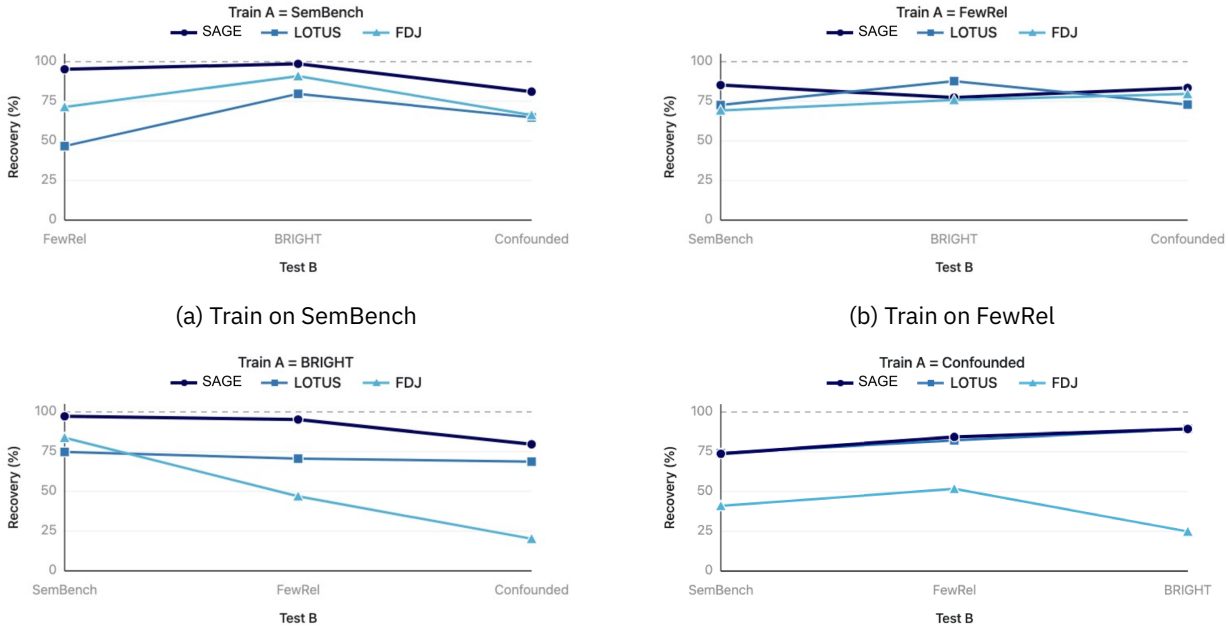
Table 6: Component ablation on one join per class (SemBench, FewRel, BRIGHT); costs in USD, higher F1 is better. Every mechanism carries weight; removing routing is the largest failure. The mean-log-probability row retains escalation and changes only the confidence signal.

[2024], Zeighami et al. [2025]; SAGE guards these choices by predicate structure, retaining pair semantics when a join cannot factor and distinguishing exact mergeable aggregates from approximate compaction. Model cascades similarly route easy inputs to cheaper models Chen et al. [2023], Ong et al. [2024]. SAGE combines item-level answer-relevance confidence Duan et al. [2024] with a query-local probe that selects and freezes a complete physical configuration. This outer search is related to iterated racing and cross-entropy methods López-Ibáñez et al. [2016], Rubinstein [1999], De Boer et al. [2005], but its candidates are semantics-preserving plans whose feasibility jointly reflects quality, cost, and latency without deployment labels.

7 Discussion, Limitations, and Conclusion

Discussion. For the audited single-pass, query-time AI functions, SAGE replaces one logical operator per surface API with three model-invocation signatures: row to value, group to value, and pair to predicate. These signatures impose the correctness obligations for physical planning: preserve row identity for `AI_SCALAR`, preserve the group result under an explicit merge or approximation contract for `AI_AGG`, and preserve pair truth for `AI_JOIN` unless the predicate is shown to factor through per-row state. The API and intent audits support the breadth of this mapping, Table 3 tests end-to-end behavior across primitive classes, and the repeated-run study provides mechanism-level evidence for predicate-aware routing, decomposition, blocking, and racing on the evaluated joins. Because the full-suite and updated join columns use disclosed but different model stacks, comparisons remain valid within each workload and protocol rather than as hardware-normalized ratios across columns.

Limitations and conclusion. The three-way coverage result is empirical, and excludes recursive agents, training, side effects, raw multimodal conversion, and index construction. Scale results cover two membership joins, while



(a) Train on SemBench (b) Train on FewRel

(c) Train on BRIGHT (d) Train on Confounded

Figure 9: Cross-domain recovery. Each panel fixes the source domain and evaluates the three held-out target domains; higher is better. No target labels are used for SAGE configuration selection.

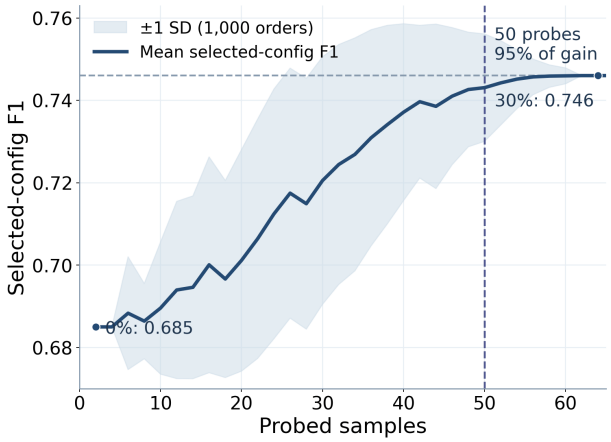


Figure 11: Configuration-search convergence for `AI_JOIN`. About fifty probed samples stabilize the selected configuration; deployed F1 rises from 0.685 to 0.746.

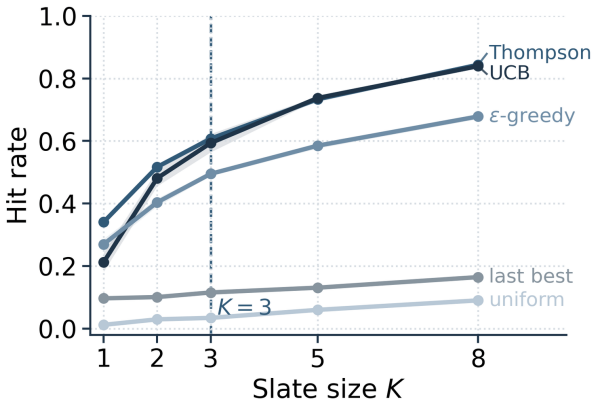


Figure 10: Near-oracle slate hit rate versus slate size K on the 160-query replay suite (mean $\pm$ s.e., 20 seeds). At the deployed K=3, the proposer places a near-oracle configuration in the slate 60.8% of the time; a uniform slate, 3.4%.

transfer results measure configuration recovery with an unlabeled target probe rather than universal task transfer. The reference-free recipe compiler and self-judged proxy may make correlated errors; aggregate compaction remains approximate without an equivalence contract; and probing must be amortized over enough inputs. Broader scalar and aggregate ablations, additional models, and production concurrency are therefore needed. Within this scope, SAGE achieves the best or tied-best updated-join quality except on Q5, replaces 16,256 pairwise calls with 128 on a representative factorable join, and reduces measured cost by 358×. The result is a small logical interface with guarded, primitive-specific execution—not a claim that all primitives share one implementation—whose richest current specialization is `AI_JOIN`.

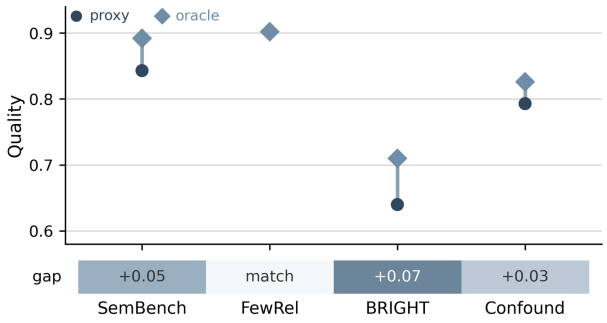


Figure 12: Quality of the proxy-selected and gold-selected configurations under the same candidate sets and budgets. Points and stems show the family-level gap; the gold selector is a diagnostic upper bound.

References

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators: a declarative model for rich, ai-based data processing. *arXiv preprint arXiv:2407.11418*, 2024.

Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. A declarative system for optimizing ai workloads. *arXiv preprint arXiv:2405.14696*, 2024a.

Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *arXiv preprint arXiv:2410.12189*, 2024.

Databricks. AI functions on Databricks. Databricks Documentation, 2025. https://docs.databricks.com/aws/en/large-language-models/ai-functions.

Snowflake. Snowflake Cortex AISQL (ai functions). Snowflake Documentation, 2025. https://docs.snowflake.com/en/user-guide/snowflake-cortex/aisql.

Google Cloud. Generative AI in BigQuery: AI.GENERATE and ML inference functions. Google Cloud BigQuery Documentation, 2025. https://cloud.google.com/bigquery/docs/generative-ai-overview.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

Jinhao Duan, Hao Cheng, Shiqi Wang, Alex Zavalny, Chenan Wang, Renjing Xu, Bhavya Kailkhura, and Kaidi Xu. Shifting attention to relevance: Towards the predictive uncertainty quantification of free-form large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5050–5063, 2024.

Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. Sembench: A benchmark for semantic query processing engines. *arXiv preprint arXiv:2511.01716*, 2025.

Yao Lu, Yue Dong, and Laurent Charlin. Multi-xscience: A large-scale dataset for extreme multi-document summarization of scientific articles. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pages 8068–8074, 2020.

Xu Han, Hao Zhu, Pengfei Yu, Ziyun Wang, Yuan Yao, Zhiyuan Liu, and Maosong Sun. Fewrel: A large-scale supervised few-shot relation classification dataset with state-of-the-art evaluation. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 4803–4809, 2018.

Hongjin Su, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han-yu Wang, Liu Haisu, Quan Shi, Zachary Siegel, Michael Tang, et al. Bright: A realistic and challenging benchmark for reasoning-intensive retrieval. In *International Conference on Learning Representations*, volume 2025, pages 48941–48991, 2025.

E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972. URL https://api.semanticscholar.org/CorpusID:41445196.

Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29:699–717, 1982. URL https://api.semanticscholar.org/CorpusID:38402199.

Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. Beyond quacking: Deep integration of language models and rag into duckdb. *arXiv preprint arXiv:2504.01157*, 2025.

Saehan Jo and Immanuel Trummer. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

Shicheng Liu, Jialiang Xu, Wesley Tjangnaka, Sina Semnani, Chen Yu, and Monica Lam. Suql: Conversational search over structured and unstructured data with large language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 4535–4555, 2024b.

Hanjun Dai, Bethany Y Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya M Phothilimthana, Charles Sutton, and Dale Schuurmans. Uqe: A query engine for unstructured databases. *Advances in Neural Information Processing Systems*, 37:29807–29838, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Ihor Stepanov, Mykhailo Shtopko, Dmytro Vodianytskyi, Oleksandr Lukashov, Alexander Yavorskyi, and Mykyta Yaroshenko. Gliclass: Generalist lightweight model for sequence classification tasks. *arXiv preprint arXiv:2508.07662*, 2025.

Jack Boylan, Chris Hokamp, and Demian Gholipour Ghalandari. Glirel-generalist model for zero-shot relation extraction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8230–8245, 2025.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1-2):484–493, 2010.

Jannis Vamvas and Rico Sennrich. X-stance: A multilingual multi-target dataset for stance detection. *arXiv preprint arXiv:2003.08385*, 2020.

David Wadden, Shanchuan Lin, Kyle Lo, Lucy Lu Wang, Madeleine van Zuylen, Arman Cohan, and Hannaneh Hajishirzi. Fact or fiction: Verifying scientific claims. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7534–7550, 2020.

IBM Granite Team. Granite 3.3 8B Instruct Model Card. https://huggingface.co/ibm-granite/granite-3.3-8b-instruct, 2025. Accessed 28 July 2026.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

I Granite Team. Granite 3.0 language models. *URL: https://github. com/ibm-granite/granite-3.0-language-models*, 2024.

Kangkang Qi, Dongyang Xie, Wenbo Li, Hao Zhang, Yuanyuan Zhu, Jeffrey Xu Yu, and Kangfei Zhao. Sema: A high-performance system for llm-based semantic query processing. *arXiv preprint arXiv:2603.11622*, 2026.

Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. Featurized-decomposition join: Low-cost semantic joins with guarantees. *arXiv preprint arXiv:2512.05399*, 2025.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G Parameswaran, and Eugene Wu. Towards accurate and efficient document analytics with large language models. *arXiv preprint arXiv:2405.04674*, 2024.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

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.

Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

Kelsey Borovinsky. “pitchfork’s reviews section by the numbers, 2021.

BerriAI. LiteLLM: Call all LLM apis using the OpenAI format. Software library, 2024. https://github.com/BerriAI/litellm.

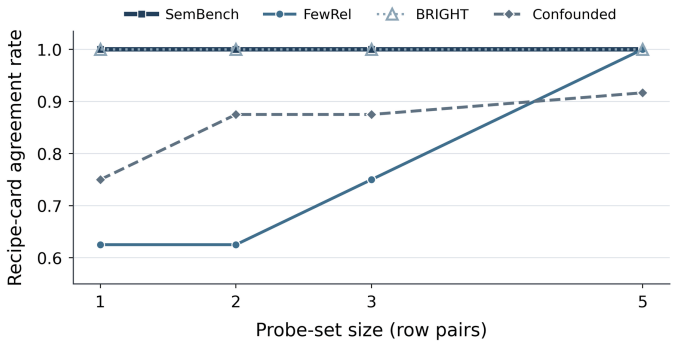


Figure 13: Recipe-card stability as the probe-set size increases. Agreement measures the fraction of generated cards matching the workload's modal five-pair structural recipe card. Small probe sets introduce structural noise for FewRel and Confounded predicates, while the generated recipes largely converge by five row pairs.

Contents of the Technical Appendix

A Recipe-Card Compiler and Routing Reliability

Figure 5 in the main paper illustrates the recipe-card compiler. For a predicate and a small `LIMIT`-style sample of at most five candidate pairs, the compiler returns the fields used by the planner: predicate type, featurizer, labels, entity labels, and comparison direction. Appendix H reproduces the corresponding prompt strings.

Because one generated card controls the route for the full join, we test both structural validity and robustness to changes in the sampled rows. Table 7 reports reference-free format checks and a sample-shift evaluation in which the predicate is held fixed while sampled rows are perturbed. The conditional "semantic correct if parseable" metric evaluates semantic routing only among successfully parsed cards; parse failures remain a separate failure mode.

Table 7: Validation of the recipe-card compiler. Conditional metrics use only the subset identified in the metric name.

Metric	Value
Reference-free validity *(no labels)*	
JSON parseable	88.3%
Schema-conformant	97.7%
Self-consistent (temp-0)	88.8%
Label well-formed	97.0%
Sample-shift robustness *(predicate fixed)*	
All-axis correct	95.7%
Semantic correct if parseable	99.7%
Sentiment path	98.3%

Route-type intervention. Syntactic validity alone does not establish that the selected execution family is semantically appropriate. Table 8 therefore changes the route while holding the evaluated query and data fixed. Removing membership factorization on SemBench increases cost, flipping sentiment direction changes the pair decisions, and removing the relational route on FewRel sharply reduces F1. On BRIGHT, substituting the relational route for the reasoning route reduces F1 and increases cost on the evaluated query. Because the route is fixed by the card and is not part of the raced configuration (Appendix E), these interventions are the only evidence we report on route choice.

Compound-predicate intervention. Table 9 separately studies a conjunction of a cheap candidate-generation condition and a semantic residual condition. Removing either condition changes the target predicate. Removing the semantic condition admits false positives; removing the cheap condition preserves more matches but eliminates pruning. Executing the whole conjunction pairwise recovers the result of the decomposed plan at a higher pairwise-call count.

A.1 Probe-Set Sensitivity of Recipe-Card Generation

We evaluate whether the size of the probe set affects recipe-card generation. For each workload, we fix the join predicate and vary only the sampled row pairs provided to the recipe-card compiler. We consider probe sets of $n \in \{1, 2, 3, 5\}$ row pairs and independently sample eight probe sets for each workload and size.

We compare recipe cards using their structural signature—operator type, featurizer, and direction—while ignoring wording differences that do not change execution. Agreement is the fraction of compilations whose structural signature matches the workload's modal five-pair recipe card. Correspondingly, 1 − agreement measures probe-induced structural variance.

Figure 13 shows that SemBench and BRIGHT are stable even with one pair. FewRel is sensitive to small probe sets, but its agreement increases from 62.5% to 100% at five pairs. Confounded predicates improve from 75.0% to 91.7%. Thus, recipe-card variance largely converges by five probe pairs: it disappears for SemBench, FewRel, and BRIGHT, with only 8.3% residual disagreement on the Confounded workloads. We therefore use five row pairs as the default probe-set size.

Dataset	Route intervention	Correct F1	Intervened F1	Cost ratio
SemBench	Membership → relational all-pairs	**0.847**	0.720	66.2×
SemBench	Sentiment direction flipped	**0.747**	0.255	1.00×
FewRel	Relational route removed	**0.902**	0.039	2.65×
BRIGHT	Reasoning route changed to relational	**0.805**	0.630	2.10×

Table 8: Route interventions on fixed diagnostic queries, not reruns of Table 3. The table isolates the effect of replacing or removing the route selected for each predicate.

Compound execution	F1 ↑	Precision	Recall	Pairwise calls ↓
Full predicate: cheap condition + semantic condition	**1.000**	**1.000**	**1.000**	286
No decomposition: whole predicate pairwise	**1.000**	**1.000**	**1.000**	505
Remove semantic/residual condition	0.754	0.605	**1.000**	**0**
Remove cheap/block condition	0.874	0.776	**1.000**	505

Table 9: Compound-predicate intervention. The valid target is the conjunction in the first two rows. The last two rows intentionally remove one condition and therefore evaluate a different predicate.

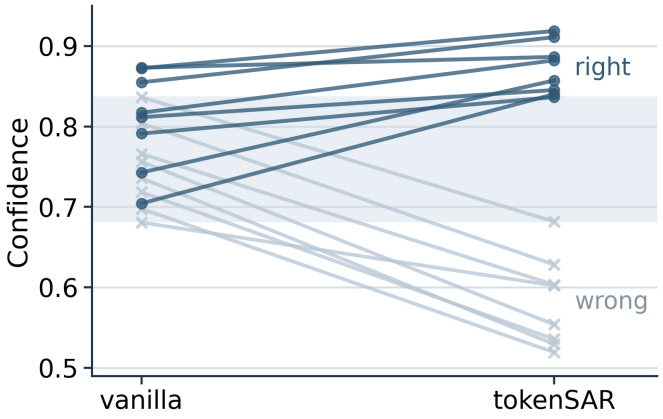


Figure 14: Per-pair confidence under mean log-probability (left) and TokenSAR (right) on the evaluated diagnostic sample. Green and red denote correct and incorrect small-model answers.

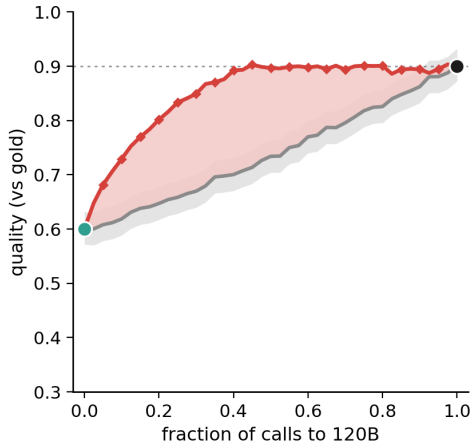


Figure 15: Confidence-ordered escalation (red) versus random escalation (grey) on the evaluated opposite-sentiment join. The dashed line is the large-model reference.

Predicate type	Dataset	GLiClass	GLiREL
Relational	FewRel	0.363	**0.842**
Reasoning	BRIGHT	0.432	**0.615**

Table 10: GLiREL versus GLiClass on balanced 100-pair diagnostic sets. Values are AUROC.

Model	Size	Join acc.
mistral-24b	24B	**0.909**
phi-4	14B	0.891
granite-8b	8B	0.873
gpt-oss-120b	120B	0.873
llama-70b	70B	0.800
qwen-72b	72B	0.764

Table 11: Per-model accuracy on the evaluated SemBench opposite-sentiment join.

B Zero-Shot Front-End Comparison

Table 10 compares GLiREL and GLiClass on hard, balanced pair sets. The comparison is a component diagnostic: it measures how well each zero-shot encoder ranks positive and negative pairs under the reported setup. It does not by itself specify the runtime assignment of front-end models.

Table 11 provides a second diagnostic on the opposite-sentiment join. Accuracy is not monotone in parameter count on this task, so the cascade's two tiers M_s and M_ℓ were chosen empirically from this measurement rather than by size. That choice is made once, offline, and holds for every join: the model pair is not part of the configuration a that the race searches (Appendix E).

C Confidence Signal and Escalation

The cascade escalates inputs for which the small-model confidence is low. Figure 14 compares mean log-probability with the TokenSAR score on the same diagnostic sample. TokenSAR is reported as a relevance-weighted uncertainty in its original formulation Duan et al. [2024]; we plot its negation so that both panels read as confidence, larger being more trusted. The plotted distributions show stronger separation for TokenSAR in this sample; the figure is a diagnostic of the confidence ordering rather than a universal calibration claim.

Figure 15 evaluates the ordering induced by confidence against randomly selecting the same number of escalations. On the opposite-sentiment join, the confidence-ordered curve reaches the large-model reference after escalating about 40% of pairs.

The saturation point differs across evaluated predicates, motivating per-query selection of the escalation threshold rather than one global threshold.

Algorithm 1 Per-query slate proposal and racing for `AI_JOIN` (Section 4)

Require: Query q with context x_q; action space $\mathcal{A}$; budgets (B_C, B_L); probe size n; slate size K; history $\mathcal{H}$
Ensure: Selected configuration $\hat{a}_q$ or Infeasible

$\{P_q^{(t)}\}_{t=1}^{T} \leftarrow$ ProbeRounds(q, n)
$\mathcal{A}_q^{\text{rem}} \leftarrow \mathcal{A}$; $\mathcal{F}_q \leftarrow \varnothing$
repeat
  $k \leftarrow \min(K, |\mathcal{A}_q^{\text{rem}}|)$
  if $\mathcal{H}$ insufficient **then**
    $S_q \leftarrow$ UniformSlate$(\mathcal{A}_q^{\text{rem}}, k)$
  else
    $\tilde{r}(a) \sim p(r(a, x_q) \mid \mathcal{H})$ for $a \in \mathcal{A}_q^{\text{rem}}$
    $S_q \leftarrow$ TopK$(\mathcal{A}_q^{\text{rem}}, \tilde{r}, k)$
  end if
  $\mathcal{R}_q \leftarrow S_q$
  for $t = 1, \ldots, T$ **do**
    for each $a \in \mathcal{R}_q$ **do**
      run a on $P_q^{(t)}$ and update $\widehat{Q}_q(a), \widehat{C}_q(a), \widehat{L}_q(a)$
    end for
    $\mathcal{R}_q \leftarrow \{a \in \mathcal{R}_q : \widehat{C}_q(a) \le B_C, \widehat{L}_q(a) \le B_L\}$
    $\mathcal{R}_q \leftarrow$ iRaceEliminate$(\mathcal{R}_q, P_q^{(1:t)})$
    if $|\mathcal{R}_q| \le 1$ **then**
      break
    end if
  end for
  for each evaluated $a \in S_q$ **do**
    $r(a, x_q) \leftarrow \widehat{Q}_q(a)\mathbf{1}[\widehat{C}_q(a) \le B_C]\mathbf{1}[\widehat{L}_q(a) \le B_L]$
  end for
  append $\{(x_q, a, r(a, x_q))\}_{a \in S_q}$ to $\mathcal{H}$; refit the posterior
  $\mathcal{F}_q \leftarrow \mathcal{R}_q$
  $\mathcal{A}_q^{\text{rem}} \leftarrow \mathcal{A}_q^{\text{rem}} \setminus S_q$
until $\mathcal{F}_q \neq \varnothing$ or $\mathcal{A}_q^{\text{rem}} = \varnothing$
if $\mathcal{F}_q = \varnothing$ **then**
  return Infeasible
end if
$\hat{a}_q \leftarrow \arg\max_{a \in \mathcal{F}_q} \widehat{Q}_q(a)$
execute the remaining $N - n$ pairs with $\hat{a}_q$
return $\hat{a}_q$

System	Approach	dang. F_1	P / R	inner F_1
SAGE (cascade)	factorize + small→large	**0.652**	0.484 / 1.00	0.278
naive-LLM	one large call per pair	0.545	0.375 / 1.00	0.03
Palimpzest	single-shot judge	0.528	0.368 / 0.93	0.00
no-skill	all-dangling	0.545	0.375 / 1.00	–

Table 12: Standalone outer-join diagnostic on the selective Pitchfork-score predicate; these values are not part of the main benchmark tables. Gold dangling rows: 15 of 40; one indicative run.

D Threshold Sensitivity

We study the local sensitivity of `AI_JOIN` to the two cascade thresholds, written τ_1 and τ_2 below ($\tau_1{=}\tau_f$ and $\tau_2{=}\tau_e$ in the main text). Figure 7 shows representative membership and reasoning surfaces in the main text; Figure 16 reports all five workloads. The arrows show the configurations visited from the default to the selected setting.

E Bandit Proposer

Algorithm 1 summarizes the per-query proposer and race. The recipe card fixes the route and cascade tiers before racing. The proposer selects a slate of at most K untested configurations, which are evaluated in rounds on the same probe. After each round, iRace removes statistically inferior or infeasible configurations. If none survives, the observations update the posterior and another slate is generated. This continues until a feasible configuration is found or the action space is exhausted.

F Proposer Replay Evaluation

The proposer is evaluated by replay, separating slate composition from the final proxy-based choice. We sample 40 queries from each of four workload families, giving 160 replay queries. For each query, all 96 configurations in $\mathcal{A}$ are executed, which supplies an observed oracle configuration for the replay analysis. A slate is counted as a hit when it contains a configuration satisfying the near-oracle criterion in Eq. 12. Curves report the mean over 20 random seeds.

Figure 17 reports three views of the replay. Panel (a) compares a history accumulated along the query stream with a cold reset. Panel (b) charges each strategy the measured cost of the probes it requests. Panel (c) varies the fraction of low-quality configurations in the replay pool. These results evaluate the specified replay stream; they do not assume that an identical history is available for a new deployment domain.

G Outer-Join Correctness and Dangling Rows

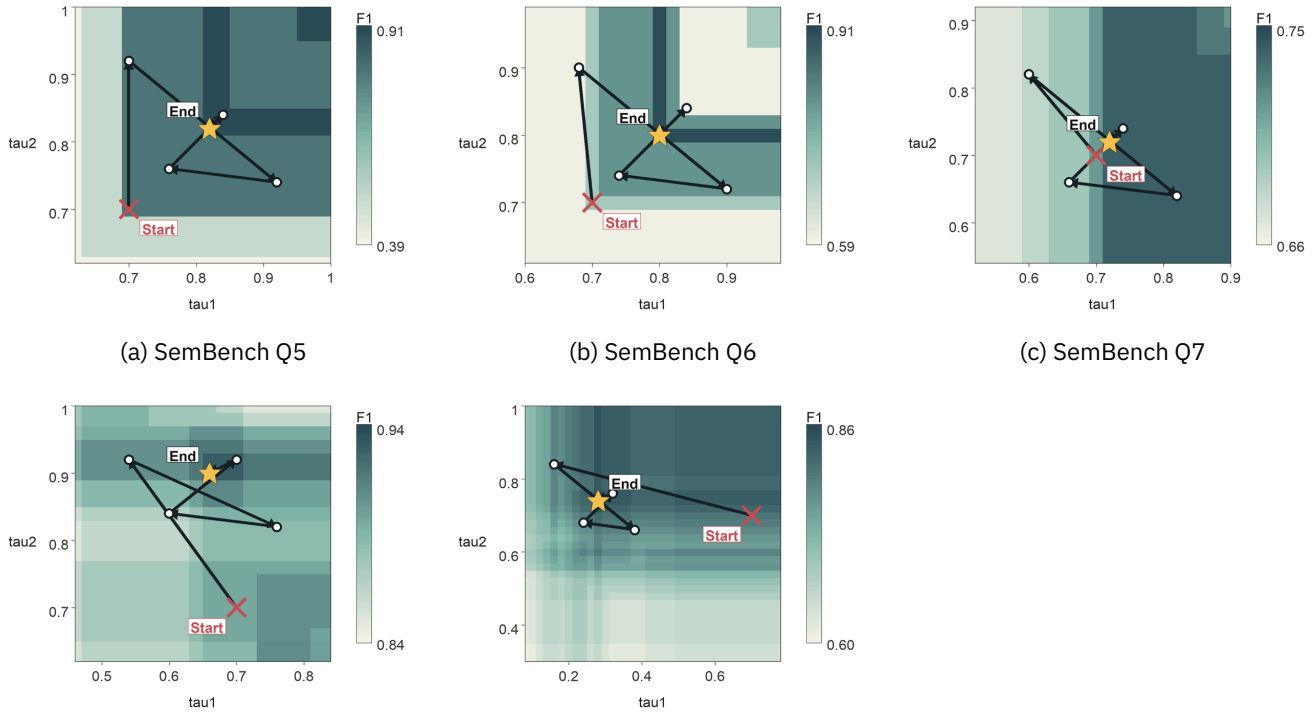
This section checks the structural behavior of the outer-join execution path. A `LEFT JOIN` must preserve each left row for which no qualifying partner is returned.

Evaluation setup. We use a selective predicate based on a whole-number Pitchfork-style score (0–10) inferred from review prose Borovinsky [2021]. The right table contains only scores {7, 8}. Under the deterministic evaluation labels, 15 of 40 left rows are dangling; the two tables produce 240 candidate pairs.

```
SELECT a.*, b.*
FROM tableA_left a
LEFT JOIN tableB_right b
  ON ai_join(a.review, b.review,
     'two reviews earn the same
      whole-number Pitchfork score (0-10)');
```

Results. Table 12 reports one run. Dangling recall measures whether the execution path preserves unmatched left rows; dangling F1 and inner F1 additionally depend on the pairwise semantic decisions.

Scope. This experiment checks outer-join structure on a small controlled input; it is not used as a general quality benchmark. Gold labels are defined for the left-side dangling-row evaluation.



(a) SemBench Q5 (b) SemBench Q6 (c) SemBench Q7

(d) FewRel (e) BRIGHT

Figure 16: AI-Join threshold sensitivity on the five evaluated workloads. Each heatmap shows the local F1 surface over the searched (τ_1, τ_2) region.

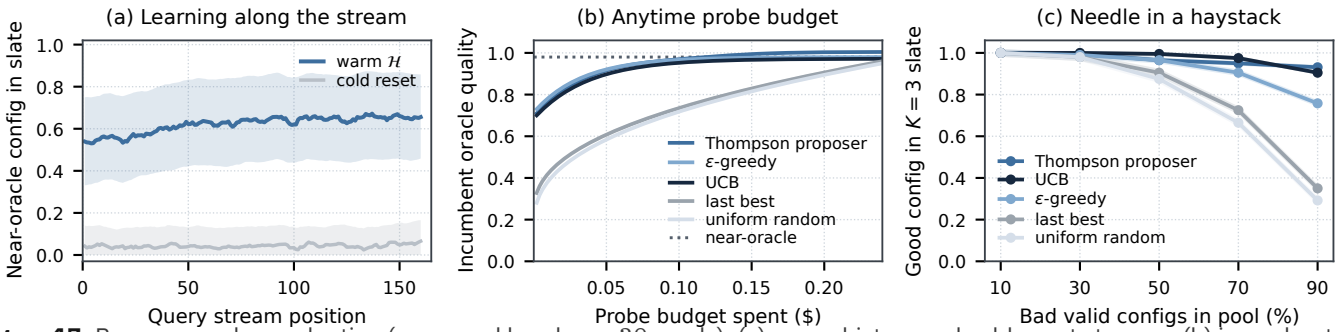


Figure 17: Proposer replay evaluation (mean and band over 20 seeds): (a) warm-history and cold-reset streams; (b) incumbent quality against measured probe cost; and (c) a controlled stress test that changes the fraction of low-quality configurations in the pool.

H Prompt Templates and Runtime Assignment

H.1 Recipe-card prompts

Card 1. Recipe-card **system** prompt

```
Compile a SQL AI_JOIN predicate into an execution recipe.
Return STRICT JSON only with:
"type": "per_side", "joint", or "reasoning".
"featurizer": "gliclass", "gliner", "numeric", "embed", or
"glirel".
"labels": a list of class or relation labels.
"entity_labels": entity types for GLiNER; otherwise [].
"direction": "same", "opposite", "similar", or "none".

Routing:
- "per_side": each row independently yields a value that is
 then compared. Use "gliclass" for closed attributes, "
gliner" for entity spans, "numeric" for numbers, and "embed
" for graded similarity.
- "joint": A and B must be scored together, without multi-
hop or external knowledge. Use "glirel".
- "reasoning": the decision requires evidence composition,
multi-hop inference, or external knowledge. Use "glirel"
only as a high-confidence negative filter; the generative
cascade decides the rest.

For "gliclass", provide mutually exclusive per-row labels.
For "glirel", provide two self-descriptive relation labels,
 with the positive label first. Leave "labels" empty for "
gliner", "numeric", and "embed". Use "direction" only for "
per_side"; otherwise use "none".

Examples:
"the two reviews express opposite sentiment" ->
{"type":"per_side","featurizer":"gliclass","labels":["
positive","negative"],"entity_labels":[],"direction":"
opposite"}

"both mention the same company" ->
{"type":"per_side","featurizer":"gliner","labels":[],"
entity_labels":["company"],"direction":"same"}

"the two entities are spouses" ->
{"type":"joint","featurizer":"glirel","labels":["the two
entities are spouses","the two entities are not spouses"],"
entity_labels":[],"direction":"none"}

"the drug in A is contraindicated for the condition in B"
->
{"type":"reasoning","featurizer":"glirel","labels":["the
drug is contraindicated for the condition","the drug is not
 contraindicated for the condition"],"entity_labels":[],"
direction":"none"}
```

Card 2. Recipe-card **user** prompt

```
Predicate: <predicate>
Example (A, B) row pairs:
  A: <left row 1>  |  B: <right row 1>
  ...(up to 5 pairs)...
Return the JSON recipe.
```

Card 3. Row-labeling **user** prompt

```
Instruction: <attr_instruction>
Allowed labels: <label 1>, <label 2>[, <label 3>]
Text: """<row text>"""
Return exactly one allowed label. If no label applies,
return none.
```

Card 4. Closed-label classification prompt

```
For the predicate "<predicate>", classify the value into
exactly one label from the closed label set. If no label
applies, return none. If the value is itself one of the
labels, return that exact label. Return only the selected
label or none.
```

Card 5. Concise model-role system prompt

```
Follow the supplied task instruction carefully. Make the
requested decision using only the provided input. Follow
the selected output-mode card exactly.
```

Card 6. Authoritative model-role system prompt

```
Act as the authoritative verifier for the supplied task.
Resolve ambiguity using the predicate and the provided
evidence, and produce the most accurate decision possible.
Follow the selected output-mode card exactly.
```

Card 7. Strict pair-judging task prompt

```
Decide whether the stated relationship holds between the
Left item and the Right item. Treat the predicate literally.
 The decision is true only if the relationship clearly
holds for this specific pair; otherwise it is false. Do not
 be lenient or speculative.
```

Card 8. Paraphrase-aware pair-judging task prompt

```
Decide whether the stated relationship holds between the
Left item and the Right item. Account for abbreviations,
synonyms, and paraphrases, but do not add unsupported facts.
 The decision is true only if the relationship clearly
holds for this specific pair; otherwise it is false.
```

Card 9. Pair-judging **user** prompt

```
Left: <left row>
Right: <right row>
Relationship to test: <predicate>

Interpret this relationship only with respect to the
supplied Left-Right pair. Determine whether it holds.
```

Card 10. Direct Boolean output mode

```
=== OUTPUT MODE: DIRECT BOOLEAN ===
Respond with exactly one lowercase word: true or false.
Do not include quotes, punctuation, explanation, or any
other text.
```

Card 11. Reasoned Boolean output mode

```
=== OUTPUT MODE: REASONED BOOLEAN ===
Give a concise, task-relevant rationale in at most three
short sentences. Then write exactly one final line in one
of these two forms:
Answer: true
Answer: false
Write nothing after the final answer line.
```

I Detailed Experimental Setup

Cross-primitive suite. Table 3 additionally reports SemBench Q1–Q4 and Q8–Q10 plus Multi-XScience. These workloads follow their standard task definitions and the quality metrics stated in the table caption. The remainder of this section documents the current repeated-run join experiments, including the compound workloads in Table 4.

Metrics and provenance. The full-suite quality scores are scaled to $[0,1]$: precision@5 for Q1–Q2, $\max(0, 1 - \text{relative error})$ for Q3, Q4, and Q8, $(\rho + 1)/2$ for the Spearman coefficient ρ on Q9–Q10, and ROUGE-1 for Multi-XScience. The full-suite SAGE execution traces identify `granite-3.3-8b-instruct` IBM Granite Team [2025] and `gpt-oss-120b` Agarwal et al. [2025]; external systems use their native execution configurations, and their token usage is priced with one shared LiteLLM schedule BerriAI [2024]. For Q5–Q7, FewRel, and BRIGHT, Table 3 replaces the earlier point estimates with the current AI-Join repeated-run means described below. Within each workload column, methods therefore share inputs and pricing; the SemBench Summary aggregates the displayed cross-primitive columns.

Table 13 summarizes the evaluated predicates and whether a Limit clause restricts the candidate set.

Join	Predicate	Limit
SemBench (Movie)		
Q5	*same sentiment*	Yes
Q6	*opposite sentiment*	Yes
Q7	*opposite sentiment*	No
FewRel	*the two entities are spouses*	No
BRIGHT	*the document supports answering the query*	No
Amazon–Google	*same product family* $\wedge$ *same underlying product*	No
x-stance	*same question* $\wedge$ *opposite stances*	No
SciFact	*same topic* $\wedge$ *citable evidence*	No

Table 13: Evaluated join workloads.

Execution settings. For every current join column, all systems evaluate the same candidate pairs under the same OpenRouter price schedule. The cascade uses `Granite-3.0-8B` as the small model and `gpt-oss-120b` as the large model. The 96 configurations combine $\tau_f \in \{0.05, 0.10, 0.15, 0.20\}$, $\tau_e \in \{0.70, 0.80, 0.85, 0.90\}$, reasoning on/off, and $T \in \{0, 0.2, 0.5\}$. Each race evaluates $K = 3$ configurations on the same uniform probe of $n = \min(50, N)$ pairs. Probe cost and latency are scaled linearly to the full candidate set and include recipe generation, racing, and judging.

Proposal and selection. We use a Bayesian linear Thompson sampler over Granite predicate representations and recipe-card classes. History is initialized with uniform slates and updated with the three observations from each query. The fixed `gpt-oss-120b` judge, run at temperature zero and blinded to configuration identity, assigns each candidate its mean probability of correctness. Default query budgets are \$0.03 and 240 seconds; if no candidate is feasible, the query reports a budget violation. Proposer evaluation uses $K = 3$ and $\epsilon = 0.05$.

J Primitive-Core Coverage and Decomposability

This bundle backs the claim that three primitives suffice: the audited map from public operator surfaces onto the primitives, the decomposition of semantic intents by composition depth, and the limits of MapReduce factorization per operator class.

J.1 Audited Operator-to-Primitive Map

We audited public semantic-query APIs—LOTUS Patel et al. [2024], Palimpzest Liu et al. [2024a], ThalamusDB Jo and Trummer [2024], DocETL Shankar et al. [2024], FlockMTL Dorbani et al. [2025], SUQL Liu et al. [2024b], UQE Dai et al. [2024], an agentic refine-until-stable loop Madaan et al. [2023], and the Databricks Databricks [2025], Snowflake Cortex Snowflake [2025], and BigQuery Google Cloud [2025] AI-SQL function suites—and mapped each exposed operator to the primitive whose documented input-output shape it implements. The audit used five annotators and reached high agreement (Fleiss' $\kappa = 0.933$). Two boundary rules matter. First, similarity, search, retrieval, vector search, and reranking are counted as `AI_JOIN` when they expose a hidden pair predicate such as "is this row relevant to the query?" Second, raw embedding construction is not a query-level semantic primitive: it is an index or blocking mechanism and is therefore marked non-semantic (NS), not `AI_SCALAR`.

J.2 Intent Decomposition by Depth

A semantic-query intent is a relational query whose values or predicates appeal to the meaning of text. Within the stated single-pass scope, every model invocation has one of three signatures, and each of the 37 audited intents decomposes into a finite plan over them: `ai_scalar` for a per-tuple judgment ($1 \to 1$), `ai_agg` for a per-group reduction ($N \to G$), and `ai_join` for a per-pair match ($N \times M \to$ pairs). Broader verbs reduce to them: a whole-relation judgment is `ai_agg` over the trivial group; search, top-k, and similarity are `ai_join` under a synthesized relevance predicate; deduplication is pairwise `ai_join` followed by relational component construction; and ranking uses either an `ai_scalar` key or query-dependent `ai_join` relevance followed by ordinary sorting. Tables 17 to 19 enumerate the audited intents: the longest chain has depth three, and most intents sit at depth one or two.

J.3 MapReduce Decomposability

Not every operator class factorizes. Table 20 classifies which plans admit an exact MapReduce decomposition and at what predicate cost. This table states algebraic execution conditions independently of the measurements in Table 3. Modular aggregation and membership joins factorize exactly, and membership joins additionally drop predicate cost to $N + M$ via per-row keys. Holistic aggregation has no exact factorization without an application-supplied merge contract. Relational and reasoning joins are MapReduce-executable but not work-reducing, since their predicates read both rows jointly and the full $N \times M$ grid must still be evaluated.

System	Operator	Documented I/O	Maps to
LOTUS	`sem_map`	per-row text → text	`AI_SCALAR`
LOTUS	`sem_filter`	per-row text → bool	`AI_SCALAR`
LOTUS	`sem_extract`	per-row text → struct	`AI_SCALAR`
LOTUS	`sem_topk`	set → ranked top-k	`AI_SCALAR` key; SQL orders
LOTUS	`sem_cluster_by`	set → cluster ids	`AI_SCALAR`
LOTUS	`sem_dedup`	set → deduped set	`AI_SCALAR`
LOTUS	`sem_join`	$N \times M$ → pairs	`AI_JOIN`
LOTUS	`sem_sim_join`	$N \times M$ → nearest pairs	`AI_JOIN` hidden predicate
LOTUS	`sem_search`	query → top-k rows	`AI_JOIN` hidden predicate
LOTUS	`sem_agg`	group → text	`AI_AGG`
LOTUS	`sem_index`	build vector index	NS: embed/index
Palimpzest	`convert`	per-row text → fields	`AI_SCALAR`
Palimpzest	`filter`	per-row text → bool	`AI_SCALAR`
Palimpzest	`join`	$N \times M$ → pairs	`AI_JOIN`
Palimpzest	`retrieve`	query → context rows	`AI_JOIN` hidden predicate
Palimpzest	`sem_agg`	group → summary	`AI_AGG`
Palimpzest	`groupby`	set → groups	NS: SQL grouping
ThalamusDB	`nlfilter`	per-row text → bool	`AI_SCALAR`
ThalamusDB	`nljoin`	$N \times M$ → pairs	`AI_JOIN`
ThalamusDB	`nl-count/agg`	group → estimate	`AI_AGG`
FlockMTL	`llm_complete`	per-row text → text	`AI_SCALAR`
FlockMTL	`llm_complete_json`	per-row text → json	`AI_SCALAR`
FlockMTL	`llm_filter`	per-row text → bool	`AI_SCALAR`
FlockMTL	`llm_rerank`	set → reranked rows	`AI_SCALAR` key
FlockMTL	`llm_reduce`	group → text	`AI_AGG`
FlockMTL	`llm_first`	group → selected row	`AI_AGG` argmax
FlockMTL	`llm_embedding`	text → vector	NS: embed
FlockMTL	`fusion_rrf`	scores → fused score	NS: SQL/scoring

Table 14: Exact operator-level map, part 1.

System	Operator	Documented I/O	Maps to
Snowflake	`AI_COMPLETE`	per-row text → text	`AI_SCALAR`
Snowflake	`AI_CLASSIFY`	per-row text → label	`AI_SCALAR`
Snowflake	`AI_FILTER (WHERE)`	per-row text → bool	`AI_SCALAR`
Snowflake	`AI_FILTER (JOIN ON)`	$N \times M$ → pairs	`AI_JOIN`
Snowflake	`AI_EXTRACT`	per-row text → struct	`AI_SCALAR`
Snowflake	`AI_SENTIMENT`	per-row text → label	`AI_SCALAR`
Snowflake	`AI_SIMILARITY`	two texts → score	`AI_JOIN` hidden predicate
Snowflake	`AI_AGG`	group → text	`AI_AGG`
Snowflake	`AI_SUMMARIZE_AGG`	group → summary	`AI_AGG`
Snowflake	`AI_EMBED`	text → vector	NS: embed
Snowflake	`AI_TRANSCRIBE`	audio → text	MM
Snowflake	`AI_PARSE_DOCUMENT`	document → text	MM
BigQuery	`AI.GENERATE`	per-row text → text	`AI_SCALAR`
BigQuery	`AI.GENERATE_BOOL`	per-row text → bool	`AI_SCALAR`
BigQuery	`AI.GENERATE_INT`	per-row text → int	`AI_SCALAR`
BigQuery	`AI.GENERATE_DOUBLE`	per-row text → double	`AI_SCALAR`
BigQuery	`AI.GENERATE_TABLE`	per-row text → rows	`AI_SCALAR` + SQL unnest
BigQuery	`AI.IF`	per-row text → bool (pair in ON)	`AI_SCALAR`
BigQuery	`AI.CLASSIFY`	per-row text → label	`AI_SCALAR`
BigQuery	`AI.SCORE`	per-row text → score	`AI_SCALAR`
BigQuery	`VECTOR_SEARCH`	query → top-k rows	`AI_JOIN` hidden predicate
BigQuery	`ML.GENERATE_EMBEDDING`	text → vector	NS: embed
BigQuery	`AI.FORECAST`	series → forecast	MM
Databricks	`ai_query`	per-row text → text	`AI_SCALAR`
Databricks	`ai_classify`	per-row text → label	`AI_SCALAR`
Databricks	`ai_extract`	per-row text → struct	`AI_SCALAR`
Databricks	`ai_filter`	per-row text → bool	`AI_SCALAR`
Databricks	`ai_gen`	per-row text → text	`AI_SCALAR`

Table 15: Exact operator-level map, part 2.

Optimizer consequences. The primitive label alone is not an optimization license. For `AI_AGG`, the optimizer may push partial aggregation into partitions only when the user-visible result is induced by a bounded mergeable state; otherwise it must preserve the group as the semantic unit. For `AI_JOIN`, distributed tiling is always an exact execution strategy, but it reduces model work only when the predicate first compiles to independent row keys. Thus "MapReduce-executable" and "model-call reducing" are distinct properties.

This distinction also determines which physical choices may be raced. Chunk size, batching, and worker placement are safe alternatives for every primitive because they preserve the logical input unit. Partial-state schemas for `AI_AGG` and per-side key extractors for `AI_JOIN` are conditional rewrites: they enter the candidate set only after the recipe proves the corresponding decomposition. The three-primitive formulation therefore narrows the optimizer's search space without pretending that every surface AI function has the same factorization law.

System	Operator	Documented I/O	Maps to
Databricks	`ai_fix_grammar`	per-row text → text	`AI_SCALAR`
Databricks	`ai_mask`	per-row text → text	`AI_SCALAR`
Databricks	`ai_summarize`	per-row text → text	`AI_SCALAR` single-row summary
Databricks	`ai_translate`	per-row text → text	`AI_SCALAR`
Databricks	`ai_analyze_sentiment`	per-row text → label	`AI_SCALAR`
Databricks	`ai_similarity`	two texts → score	`AI_JOIN` hidden predicate
Databricks	`vector_search`	query → top-k rows	`AI_JOIN` hidden predicate
Databricks	`ai_forecast`	series → forecast	MM
Databricks	`ai_parse_document`	document → text	MM
DocETL	`map`	per-doc → fields	`AI_SCALAR`
DocETL	`filter`	per-doc → bool	`AI_SCALAR`
DocETL	`cluster`	set → clusters	`AI_SCALAR`
DocETL	`resolve`	set → canonical entities	`AI_JOIN` same-entity pairs
DocETL	`equijoin`	$N \times M \rightarrow$ pairs	`AI_JOIN`
DocETL	`reduce`	group → output	`AI_AGG`
DocETL	`gather`	doc+peers → context	NS: SQL/context
DocETL	`split`	doc → chunks	NS: SQL/chunking
DocETL	`unnest`	array → rows	NS: SQL unnest
SUQL	`answer`	passages → answer	`AI_AGG`
SUQL	`summary`	column → summary	`AI_AGG`
UQE	semantic aggregation query	group → estimate	`AI_AGG`
Generic	agentic refine-until-stable	data-dependent rounds	NS: loop

Table 16: Exact operator-level map, part 3.

ID	Intent	Plan shape	Example
A01	group summarization	`AI_AGG`	summarize each group theme
A02	group theme extraction	`AI_AGG`	return recurring themes per product category
A03	semantic estimate over a group	`AI_AGG`	estimate count of complaints in each group
A04	group-level decision	`AI_AGG`	judge whether a ticket cluster has a common root cause
R01	text similarity metric	`AI_JOIN`	P(similar) for two texts via the join's pair head
R02	top-k semantic retrieval	`AI_JOIN`	retrieve top-k items relevant to a query
S01	row sentiment classification	`AI_SCALAR`	classify each review sentiment
S02	row topic or category label	`AI_SCALAR`	classify product category
S03	row boolean semantic filter	`AI_SCALAR`	flag positive reviews
S04	row validation predicate	`AI_SCALAR`	validate whether a support note is complete
S05	row numeric score	`AI_SCALAR`	score review satisfaction from 1 to 5
S06	single-field extraction	`AI_SCALAR`	extract the product name from a review
S07	multi-field JSON extraction	`AI_SCALAR`	extract product issue and severity as JSON
S08	row translation	`AI_SCALAR`	translate a customer note to English
S09	row summarization	`AI_SCALAR`	summarize one review in five words
S10	row rewrite or grammar fix	`AI_SCALAR`	rewrite a ticket as a concise title
S11	row redaction or masking	`AI_SCALAR`	redact PII from a support note
S12	row format conversion	`AI_SCALAR`	convert a free-text address to normalized JSON
S13	row explanation or enrichment	`AI_SCALAR`	explain an SAP payment term in plain English
S14	row free-text generation	`AI_SCALAR`	generate a short response draft
S15	typed generation	`AI_SCALAR`	generate a boolean or integer answer
S16	compare two supplied texts in one row	`AI_SCALAR`	compare two already paired reviews
S17	row moderation or safety label	`AI_SCALAR`	label whether a note is policy-safe
S18	row department classification	`AI_SCALAR`	classify material description into department
J01	semantic entity join	`AI_JOIN`	match product listing to catalog item
J02	evidence or claim match	`AI_JOIN`	match review to claim
J03	pairwise relation join	`AI_JOIN`	match reviews with opposite sentiment

Table 17: Depth-1 intent decompositions. Retrieval and similarity (R01, R02) are the one-sided or pairwise `AI_JOIN` under a synthesized predicate; there is no embedding operator.

ID	Intent	Plan shape	Example
D203	join then summarize	AI_JOIN → AI_AGG	join tickets to policies then summarize violations
D204	match then aggregate evidence	AI_JOIN → AI_AGG	match reviews to claims then aggregate evidence
D207	retrieve then answer	AI_JOIN → AI_SCALAR	retrieve context then answer per row
D201	classify then summarize	AI_SCALAR → AI_AGG	classify complaints then summarize by category
D202	extract then aggregate	AI_SCALAR → AI_AGG	extract issue type then count themes by region
D205	extract then match	AI_SCALAR → AI_JOIN	extract entities then match records
D206	normalize then join	AI_SCALAR → AI_JOIN	normalize product names then semantic join
D208	extract then score	AI_SCALAR → AI_SCALAR	extract field then score the extracted value

Table 18: Depth-2 intent decompositions.

ID	Intent	Plan shape	Example
D301	join judge aggregate	AI_JOIN → AI_SCALAR → AI_AGG	match docs, judge relation, aggregate evidence
D302	extract join aggregate	AI_SCALAR → AI_JOIN → AI_AGG	extract entities, join to claims, summarize matches

Table 19: Depth-3 intent decompositions.

Table 20: MapReduce decomposability by operator class. For joins, exact MapReduce tiles the full $N \times M$ pair grid and unions positive pairs. Predicate cost is the number of AI predicate evaluations required by the cheapest exact decomposition.

Class	Example predicate	Map (per-chunk)	Reduce	Exact?	Cost
Modular agg	count positive, mean, min/max	bounded partial state	add / extremum / tally	√	N
Holistic agg	median, top-k themes, summary	naive partial answer	partition-dependent merge	×	—
Membership join	same category / sentiment	per-item key label	hash-join on key	√	$N+M$
Relational join	R(head, tail)	pair-tile evaluation	union positive pairs	√	$N \times M$
Reasoning join	document answers query	pair-tile evaluation	union positive pairs	√	$N \times M$