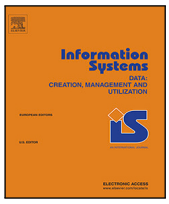
Information Systems 143 (2027) 102796

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



IOLM-DB: Instance-optimized large language model execution for column-scale semantic operators☆

Bardia Mohammadi *, Laurent Bindschaedler

Max Planck Institute for Software Systems, Germany

ARTICLE INFO

Keywords:
Semantic operators
Tabular data
Large language models
Model compression
Quantization
Knowledge distillation
Instance optimization
Efficiency

ABSTRACT

Large language models (LLMs) can implement row-level semantic transformations over tabular data, but a general-purpose model reserves the same accelerator resources even when every row uses one fixed prompt. IOLM-DB compiles that recurring prompt–column pair into an operator sized to the work it actually does. It samples the target column, constructs calibration sequences that represent both prompt and output behavior, evaluates specialized candidates, and selects an implementation under explicit memory and quality constraints. Across heterogeneous datasets and operator types, column-calibrated quantization reduces the resident footprint of the reference model by 1.8–2.8× while the 8-bit profile remains a near-lossless replacement and the 4-bit profile preserves behavior on categorical label-output operators. At fixed precision, column calibration consistently improves fidelity over generic GPTQ calibration, and output-aware calibration provides an additional benefit for generative operators. Deployment experiments reveal two complementary outcomes: weight-only compression does not raise per-row throughput in a compute-rich regime, where it primarily releases capacity, but the smaller weight stream also improves throughput when memory bandwidth becomes limiting. The compact artifacts enable deployment under a memory budget where the full-precision operator cannot start and allow three specialized operators to occupy roughly the footprint of one full-precision model. IOLM-DB therefore turns a recurring semantic transformation from an immutable call to a general-purpose backend into a compact, measurable, and hardware-aware operator.

1. Introduction

Large language models (LLMs) let analysts express row-level text transformations over tabular data, including summarization, data correction, fuzzy entity matching, and classification [1–5]. This interface brings semantic text processing into analytical workflows. At column scale, however, model execution can dominate GPU memory, compute time, and monetary cost.

Consider an analyst attaching a sentiment label to every row of a five-million-row product-reviews table, or normalizing a free-text address column before joining two supplier lists. Each task is a single fixed instruction applied row by row. Routing five million such calls to a hosted application programming interface (API) accrues per-token charges and network latency that grow linearly with the table. Serving a general-purpose 8B model locally removes the per-call charge, but it reserves enough GPU memory to crowd out concurrent work. In both cases the general-purpose model carries capacity (multilingual translation, code synthesis, multi-turn dialogue) that a single labeling operator never exercises. This paper asks what is gained by specializing the model to the operator before running it at column scale.

Today there are two ways to execute these operators: through a remote API, or through a general-purpose model served locally. LOTUS [6] exposes semantic operators over DataFrames, FlockMTL [7] embeds model-driven functions in DuckDB [8], and Palimpzest [9] optimizes declarative pipelines over model-powered operators. These systems improve expression, planning, batching, and caching. The model artifact behind each operator remains unchanged, so every operator retains the general model's full resource footprint.

This paper studies a focused optimization problem: given a fixed prompt template and a target column drawn from a roughly stationary distribution, can a reference LLM be specialized into a smaller operator that preserves the behavior required by that workload? IOLM-DB treats the prompt and target column as one optimization instance. It samples the column, constructs calibration sequences that represent both prompt and output behavior, evaluates compressed candidates, and deploys the candidate that satisfies the workload's quality and

☆ This article is part of a Special issue entitled: 'DOLAP 2025' published in Information Systems.

* Corresponding author.
E-mail address: bmohammadi@mpi-sws.org (B. Mohammadi).

https://doi.org/10.1016/j.is.2026.102796
Received 2 September 2025; Received in revised form 12 August 2026; Accepted 19 August 2026
Available online 1 September 2026

memory constraints. A recurring operator over one column exercises only a narrow slice of the reference model's general-purpose behavior, creating an opportunity for workload-specific resource sizing.

We evaluate IOLM-DB on three semantic-operator workloads: summarization over Amazon Reviews [10], data correction over the GitHub Typo Corpus [11], and fuzzy entity join. We compare against local per-row inference, batched local inference, GPT-4o API calls as a cloud reference, and a distilled small-model baseline.

Column-calibrated compression shrinks the resident Llama-3.1-8B operator from 14.98 GB to 8.46 GB at 8-bit and 5.31 GB at 4-bit. The 8-bit variant preserves 0.93–1.00 of reference behavior across the evaluated workloads. At 4-bit, fidelity is 0.96–0.98 on Movies and Products Filter, 0.91 on BIRD Filter, and 0.61–0.67 on generative Projection, motivating separate *Fidelity* and *Compact* deployment profiles. What happens to throughput depends on the hardware and on the fidelity the workload demands. On the 80 GB A100, the high-fidelity weight-only profiles mainly release capacity, while a quality-relaxed W8A8 candidate reaches 1.41× throughput. On the lower-bandwidth L40S, Compact raises throughput by 1.28× and serves under a 14.8 GB cap where FP16 cannot start (Section 5.5). The smaller artifact also enables three distinct operators to co-reside in 15.93 GB and supports denser provisioning. Batching remains a complementary throughput lever, while distillation reduces the footprint to 0.23 GB for constrained operators.

This paper makes column-calibrated footprint reduction the primary optimization for column-scale semantic operators. Specializing the operator's model to its column shrinks the resident artifact by up to 2.8× at controlled fidelity, changes the hardware class on which it can be deployed, and increases operator density per device. A cost-based planner then selects the appropriate artifact for each deployment. Concretely, we make three contributions.

- We define instance optimization for a fixed LLM semantic operator over a target column and show that column-calibrated quantization reduces its resident footprint by 1.8–2.8×, with near-lossless fidelity at 8-bit and on the Movies and Products Filter operators at 4-bit.
- We introduce column- and output-aware calibration for semantic operators, specify a measured per-layer bitwidth procedure, and quantify the incremental benefit over column-agnostic GPTQ: column calibration improves all six workload–operator cells, and output-aware calibration improves every generative cell (Section 5.3).
- We establish the deployment consequences across heterogeneous workloads, three model architectures, a DataFrame engine, and two GPU regimes. We show a 1.28× hardware-dependent throughput gain, three-operator co-residency within the footprint of one FP16 model, and CBOP decisions that match the measured oracle across the evaluated regimes (Section 5.6).

2. Background and motivation

2.1. Semantic operators over tables

An LLM semantic operator is a fixed natural-language transformation applied to each row of a table or to each candidate pair in a join. Unlike conventional scalar code, the operator executes a model with a prompt template and row values as inputs, so its cost depends on token length, model size, batching policy, and output length.

Database and DataFrame systems expose these operators through SQL functions, user-defined functions (UDFs), or DataFrame methods. This interface is useful because it keeps semantic text processing close to filtering, joining, and aggregation. It also creates a cost problem: a query that touches N rows may invoke the model N times unless the execution engine can batch, cache, reorder, or avoid calls.

Why row-by-row invocation does not scale. Remote API execution has two costs that scale with cardinality: per-token charges and network round trips. Local execution avoids per-call charges, yet a billion-parameter model still consumes GPU memory and scheduler capacity even when each row performs a small transformation. Batching and serving optimizations generally improve hardware utilization, but the resident model footprint remains unchanged, and batching gains saturate when prompts or outputs are short.

Expensive predicates and UDFs have long been studied in the database community, where optimizers reorder them using cost and selectivity estimates [12]. Those techniques still matter for LLM operators. They reduce avoidable calls, while systems must still reduce the cost of each remaining model invocation. A model-backed UDF therefore needs both query-level optimization and model-level cost reduction when the same operator runs over a large column.

Model-side optimization tools. Existing model-optimization techniques provide the low-level tools. Post-training quantization methods such as GPTQ [13] and SmoothQuant [14] lower numerical precision to reduce memory traffic and storage. Pruning methods such as SparseGPT [15] and LLM-Pruner [16] remove weights or transformer components with low measured importance. Knowledge distillation transfers behavior from a large teacher model to a smaller student model [17,18].

These techniques usually target broad model utility across many prompts and domains. Tabular semantic operators expose a narrower setting: the prompt template is fixed, and the target column is a finite distribution that can be sampled before full execution. That setting gives the optimizer information that broad-domain compression does not use. It also raises a stricter correctness question: the optimized operator must remain predictable enough to run inside structured query execution.

The gap. Existing tabular LLM systems make semantic operators easier to express and schedule, while existing compression methods make general models smaller or faster. The missing case is the repeated, fixed semantic operator over a known column distribution. In this case, the system can sample the exact inputs the operator will see, estimate whether one-time optimization cost will amortize, and decide how much reference-model fidelity the workload requires.

For this setting, the system needs a calibration set that covers common rows and recurring outliers, a measurable quality target for compression or distillation, and a safe execution path for rows that fall outside the calibrated distribution.

Terminology and scope. We use *LLM operator* to mean one well-defined text transformation applied row by row within a tabular pipeline. We use *instance-optimized* to mean specialized for a specific prompt template and target distribution, not for each ad hoc query string. IOLM-DB specializes one operator per optimization pass, and the resulting artifacts can be composed in multi-operator plans, as demonstrated by the co-residency experiment in Section 5.5.

3. System design and architecture

IOLM-DB specializes a general-purpose LLM into a compact, task-specific *semantic operator* for a target tabular column. The engine-agnostic pipeline takes a prompt template, a reference LLM, and a sample of the target column, then returns a smaller operator with measured fidelity and hardware-specific execution cost.

3.1. Overview and design principles

The design relies on a fixed prompt template and a column whose row distribution is reasonably stable. It also assumes the column is large enough that a one-time optimization step can be amortized over thousands to millions of model calls.

The *Calibration Sampler* draws a representative subset of rows from the target column. The *Model Optimization Engine* uses that sample

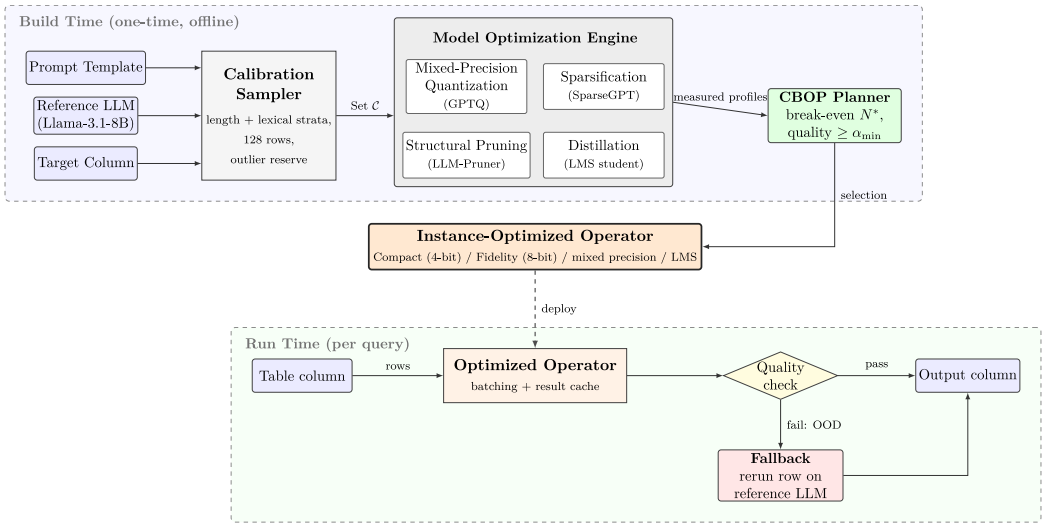


Fig. 1. IOLM-DB architecture. **Build time:** the Calibration Sampler produces $\mathcal{C}$ from the prompt and target column; the Model Optimization Engine evaluates quantized, sparse, pruned, and distilled candidates; CBOP selects the deployment recipe. **Run time:** the selected operator executes with batching and caching, while guardrail violations trigger row-level fallback to the reference model.

to build and evaluate quantized, sparse, pruned, and distilled candidates. CBOP selects a candidate that satisfies the deployment's quality and memory constraints. At runtime, IOLM-DB invokes the selected operator with optional batching and result caching, then reports per-row outputs. Fig. 1 separates this one-time build path from per-query execution.

IOLM-DB uses the prompt and sampled rows directly when choosing optimization settings instead of relying on a generic corpus. This column-specific signal improves fidelity at a fixed bitwidth and footprint (Section 5.3).

3.2. Techniques for generating instance-optimized models

The optimization search space includes model compression, invocation changes, and distillation. Candidate construction uses the target column for calibration. The *Calibration-Set Selection* and *Per-Layer Sensitivity and Bitwidth Profiles* paragraphs below describe the two choices that most affect the resulting operator: calibration-set construction and per-layer bitwidth assignment.

3.2.1. Model-level optimization

Model-level compression changes the weights or structure of the reference LLM to reduce the operator's memory footprint and, when the serving kernel can exploit it, execution cost. Because the prompt and target column are fixed, candidate selection can optimize for their measured behavior instead of broad-domain utility.

Quantization. We use post-training weight quantization (GPTQ [13]), optionally combined with SmoothQuant-style [14] activation scaling to control activation outliers. The search space contains uniform W4A16 and W8A16 candidates, a W8A8 activation-quantized candidate, and a mixed-precision candidate whose transformer sub-layer bitwidths follow the sensitivity procedure in Section 3.2.1. Lower precision reduces weight storage and resident GPU memory, while its throughput effect depends on the serving kernel and the hardware balance. Across our workloads, W8A16 is near-lossless, W4A16 is strongest on categorical Filter operators, and W8A8 provides a throughput-oriented point under a relaxed fidelity floor.

Sparsification. We evaluate 50% unstructured sparsity with SparseGPT [15], calibrated on the same sample as quantization. Zeroing weights reduces the number of nonzero parameters, but the evaluated artifact benefits only if the serving stack exposes a compatible sparse kernel. Hardware-supported 2:4 sparsity is a distinct future candidate and is not evaluated here.

Structural pruning. Structural pruning removes transformer components, specifically attention heads and feed-forward layers, whose contribution to the operator's output on the calibration set is small. We follow LLM-Pruner [16]: we compute gradient-based importance scores for each head and layer with respect to the reference model's outputs on the calibration sample, and remove components whose aggregate importance falls below a per-type threshold. Because the scoring uses the target column, pruning follows this operator's behavior rather than general-purpose benchmark performance.

The candidate builder can compose structural pruning, sparsification, and quantization in that order, using the same calibration sample throughout. It also evaluates each technique in isolation. Validation then determines which stages survive into the deployed artifact. In the current serving stack, the selected Compact and Fidelity artifacts are quantization-only variants.

What survives into a servable artifact, and what does not. The optimizer searches over all three compression families, but the ablation in Section 5.3 identifies quantization as the production path in the current serving stack. Structural pruning preserves fidelity but barely reduces resident memory, while unstructured sparsification lacks an effective accelerated serving path. The deployed Compact and Fidelity profiles therefore use GPTQ, augmented by IOLM-DB's column- and output-aware calibration, sensitivity measurement, validation, and planning. This distinction defines the contribution precisely. IOLM-DB does not replace the underlying quantizer. It turns quantization into a column-specific procedure for compiling an operator. Section 5.3 measures its incremental gain over generic GPTQ at identical precision and footprint.

Calibration-set selection. The calibration set matters because the compression algorithms only observe those rows from the target column: they estimate each layer's quantization error from forward passes over $\mathcal{C}$ alone. What the operator is calibrated *on* is therefore the one degree of freedom that instance optimization controls, and we specify it explicitly.

Which rows. We construct $|\mathcal{C}| = 128$ under a fixed seed. The sampler divides the column into input-length quartiles, allocates rows across k-means lexical clusters within each quartile, and reserves eight positions for the longest rows and the least-populated lexical cluster. Sampling remains uniform within each stratum. This procedure makes coverage deterministic for a prompt–column pair and prevents common short rows from consuming the full budget. We also compare it with

a simple uniform column sample. Across all three datasets and both operators, the fidelity difference remains within the run-to-run variance reported in Table 5. We retain stratification to guarantee coverage. The factors that measurably improve fidelity are the calibration *source* (column rather than generic text) and, for generative operators, the reference outputs we append to each sequence.

Input-only versus output-aware. By default calibration is unlabeled: each sequence is the operator's prompt applied to a sampled row, and compression needs only that input distribution. This is sufficient for constrained operators, whose output is a single token. For generative operators it is not, because a prompt-only sequence ends before the model generates, so the decode-phase activations the operator actually spends most of its compute on are absent from the calibration statistics. When the operator is generative we therefore use an *output-aware* calibration set: we append the reference model's completion on each calibration row as the sequence tail, so the calibration statistics include the decode phase. This consumes the reference model's outputs on the 128 calibration rows, which the pipeline already produces when it establishes the baseline, so it adds no inference, and is possible only because the operator is fixed and known, which a generic corpus is not. Its measured effect is confined to generative operators (Table 5).

For the distillation pipeline (Section 3.2.3), the same stratified sampler draws the rows that are teacher-labeled to form the student's training set, and the optimizer retains the reference model's outputs as training targets. The distilled-student experiments use a 100-row teacher-labeled training budget (Table 8).

Per-layer sensitivity and bitwidth profiles. Mixed-precision quantization depends on identifying layers that tolerate low precision. Uniform 4-bit quantization can sacrifice avoidable fidelity, while uniform 8-bit quantization leaves footprint reduction on the table. We measure a sensitivity-ranked profile as follows:

1. **Per-layer sensitivity score.** For each transformer sub-layer ℓ (each attention projection W_Q, W_K, W_V, W_O and each feed-forward matrix $W_{\text{gate}}, W_{\text{up}}, W_{\text{down}}$, for 224 sub-layers in Llama-3.1-8B), we measure $\Delta_\ell = \mathbb{E}_{x\sim\mathcal{C}} \|f_{\text{ref}}(x) - f_{\text{quant}(\ell,4)}(x)\|_2$. The probe quantizes only layer ℓ with groupwise round-to-nearest precision and leaves all other layers unchanged. This inexpensive perturbation ranks intrinsic layer sensitivity; the final artifact is built with GPTQ.
2. **Sensitivity-ranked profile.** We sort layers by Δ_ℓ. The top one-third receive 8-bit precision, and the remainder receive 4-bit precision. This profile maximizes fidelity without an explicit footprint budget.
3. **Outlier-aware candidate.** A SmoothQuant candidate [14] redistributes activation outliers before quantization. CBOP retains it when held-out validation improves at the target footprint.
4. **Embedding and language-model head.** We keep the token embedding and final language-model output projection at 8-bit precision, because perturbations there directly affect the output distribution.

The procedure is deterministic given the calibration set and reference model. The optimizer stores both the measured scores and the resulting map, allowing CBOP to generate alternative bitwidth profiles for different memory budgets.

Measured sensitivity structure. We run the probe on Movies, Products, and BIRD for the Projection operator with 64 calibration rows. The feed-forward read-out matrix W_{down} is the most sensitive by a wide margin (mean $\Delta_\ell = 24.4$ and membership in the 8-bit set for 28–32 of 32 layers per column), followed by W_{up} (18.7) and the attention value projection W_V (17.7). The query and key projections W_Q and W_K are least sensitive (8.4 and 6.8) and never enter the top third. This ordering agrees with the activation-outlier mechanism targeted by SmoothQuant and AWQ [14,19]: W_{down} consumes the outlier-heavy intermediate activations. The three 8-bit sets overlap by 75%–81%, showing that architecture determines most of the ordering while the column resolves the remaining assignments. The map is descriptive evidence about where perturbations concentrate; we do not evaluate the sensitivity-ranked profile itself as a served checkpoint. Because the largest matrices are also the most sensitive, that profile would approach the fully 8-bit footprint. Section 5.3 instead evaluates one footprint-biased profile that keeps the smaller attention block at 8-bit and the parameter-heavy feed-forward matrices at 4-bit. The stored scores support constructing other budget-specific profiles, but their serving behavior requires separate validation.

Calibration and compression protocol. The optimizer follows a fixed protocol:

1. **Sampling:** extract C using the procedure above.
2. **Candidate generation:** build candidate variants over a predefined search space (quantization-only, sparsity-only, pruning-only, and combined variants).
3. **Validation:** evaluate each candidate on a held-out validation slice and record fidelity and task-quality deltas relative to the reference model.
4. **Selection:** choose the feasible candidate with the lowest predicted end-to-end cost under the user's quality and memory constraints.
5. **Fallback registration:** register a local or remote reference endpoint for rows that fail runtime quality checks.

The planner uses these measurements to distinguish the effect of individual techniques from the effect of combined variants.

3.2.2. Invocation-level optimization

Batching groups rows to amortize model invocation overhead. IOLM-DB sets batch size dynamically using query latency requirements, available GPU memory, and concurrent workload demand.

Caching exploits locality in analytical workloads. The cache stores duplicate or near-duplicate inputs, intermediate representations for common prompt patterns, and final results for identical transformations.

Row-level fallback. Because compression is calibrated against a finite sample, individual rows may fall outside the regime where the compressed operator matches the reference model. The runtime therefore applies a configurable quality check to each output (empty response, pattern mismatch, or length-based anomaly). When the check fails, the runtime routes the row to a local or remote reference endpoint and logs the event. This fallback contains detectable out-of-distribution failures, and its event log feeds the recalibration trigger discussed in Section 7.

3.2.3. Architecture-level optimization

Task-specific distillation transfers reference behavior to smaller models [17,18]. Lightweight Model Synthesis (LMS) builds on ScaleLLM [20]. It applies a teacher to representative rows from the target column, eliminating manual annotation and producing lightweight models for the operator.

LMS extends ScaleLLM's classification-oriented workflow to accept both categorical and generative teacher targets. It samples rows with the stratified procedure in Section 3.2.1, runs the reference LLM to label them, and fine-tunes a T5-Small student on the resulting (input, teacher-output) pairs. CBOP admits the student only when held-out fidelity meets $\alpha_{\min}$.

3.3. Planner decision space

For a fixed prompt and target column, IOLM-DB can invoke the reference model directly, batch calls, compress the reference model, use a native small model, or distill the task into a smaller student.

These options trade reference-model fidelity, resident footprint, per-row throughput, and one-time generation cost. Direct invocation avoids artifact-generation overhead. Batching amortizes invocation overhead without changing the model. Compression reduces resident memory and can raise throughput when weight bandwidth binds. Native small models and distilled students offer still smaller or faster operators when they meet the fidelity threshold. External APIs such as GPT-4o sit outside this local design space and provide a monetary-cost reference whose per-token charges and network latency scale with column cardinality.

The best strategy depends on operator type, quality target, and hardware. The 4-bit profile minimizes footprint and preserves categorical Filter behavior, while the 8-bit profile protects generative fidelity. On the A100, batching is the high-fidelity throughput choice, and W8A8 becomes the throughput choice under a relaxed fidelity floor. On the L40S, 4-bit weight-only compression provides both capacity and speed. Native 3B models lead Filter throughput but lose generative fidelity, and LMS is effective when the target output space is sufficiently constrained. The next subsection formalizes these choices using measured generation cost, row throughput, fidelity, and memory fit.

3.4. Cost-based optimization planner

The *Cost-Based Optimization Planner (CBOP)* selects among direct invocation, batching, quantized variants, native small models, and distilled students. For every strategy s, profiling records generation time G_s, throughput r_s, fidelity a_s, and resident footprint m_s. Given available memory M and a minimum fidelity $\alpha_{\min}$, CBOP first constructs the feasible set

$$\mathcal{F}(M, \alpha_{\min}) = \{s \mid m_s \leq M \ \wedge \ a_s \geq \alpha_{\min}\}. \tag{1}$$

The fidelity floor is a deployment policy rather than a universal constant. The UDF exposes a conservative default of 0.95; the evaluation also reports 0.90 as a replacement-sensitive setting and 0.75 as an explicitly quality-relaxed setting. Where labeled validation data exist, operators can set the floor from absolute task accuracy; otherwise it expresses the maximum acceptable divergence from the reference behavior. Here M is the artifact-residency budget remaining after the serving runtime reserves memory for its context, captured graphs, and key–value cache. Hardware profiling performs that conversion from physical device capacity; an artifact smaller than the raw device capacity is not automatically serviceable. It then predicts the end-to-end operator time for N rows,

$$T_s(N) = G_s + \frac{N}{r_s}, \tag{2}$$

and selects $s^* = \arg\min_{s \in \mathcal{F}} T_s(N)$. Direct invocation and batching have $G_s = 0$, whereas generated artifacts include their measured calibration, validation, and serialization costs. A co-residency request replaces the single-artifact memory test with the sum of the concurrently resident footprints.

For comparison with a baseline strategy b, the break-even cardinality of candidate s is

$$N^*_{b \to s} = \frac{G_s - G_b}{\left(\frac{1}{r_b} - \frac{1}{r_s}\right)}. \tag{3}$$

When $r_s \leq r_b$, the throughput break-even is unbounded, and CBOP selects s only when its footprint makes the baseline infeasible or when co-residency is required. When $r_s > r_b$, the same equation yields a finite row threshold. This single rule captures both measured hardware regimes: weight-only Compact is a capacity choice on the A100 and a throughput-and-capacity choice on the L40S.

At full-query scope, CBOP combines this operator-level model with relational execution costs:

$$T_{\text{query}} = T_{\text{scan/join/agg}} + T_{\text{llm-ops}}. \tag{4}$$

Because $T_{\text{scan/join/agg}}$ is common to baseline and optimized plans for the same relational plan, CBOP focuses its decision on $T_{\text{llm-ops}}$, while using optimizer-estimated post-filter cardinalities to avoid naive row-count assumptions.

To make planner behavior auditable, we decompose each artifact's generation cost:

$$G_s = T_{\text{sample}} + T_{\text{candidate}} + T_{\text{validate}} + T_{\text{serialize}}. \tag{5}$$

CBOP logs the decomposition for each generated candidate and later uses it to explain amortization behavior in the evaluation.

The planner first records the row count, token profile, and recurrence estimate. It then projects throughput and quality for the available strategies and checks parallelism and memory fit. These logged quantities make the optimize-vs-direct decision explicit and testable.

4. Workloads and use cases

We evaluate IOLM-DB on row-level semantic-operator workloads chosen to stress different parts of the trade-off space defined in Section 3.3: long generative output, constrained correction, and short binary classification. Each workload applies one fixed prompt template across a column or candidate-pair table, matching the stationarity assumption in Section 3.

- **Summarization over Amazon Reviews.** The operator condenses free-text product reviews from the Amazon Reviews corpus [10] into short summaries. This workload stresses generative output and longer input sequences.
- **Data correction over GitHub Typos.** The operator rewrites candidate typo-containing text from the GitHub Typo Corpus [11] while preserving meaning. This workload tests constrained generation where small semantic changes are failures.
- **Fuzzy entity join.** The operator receives a candidate pair of entity descriptions and returns whether the pair refers to the same entity. This workload tests short inputs and constrained yes/no output.

Together, these three workloads cover the principal output regimes a tabular semantic operator exhibits: free-text generation, constrained generation, and binary classification.

Intended deployment scenarios. IOLM-DB targets recurring semantic operators whose deployment is constrained by memory, operator density, or memory bandwidth. First, *co-residency on a shared analytics server* allows a sentiment filter, category projection, and address normalizer to remain loaded at the same time, since three Compact operators occupy 15.93 GB, approximately the footprint of one FP16 operator (Section 5.5). Second, *lower-memory deployment* moves an operator into a 16 GB-class budget: under a measured 14.8 GB cap, both compressed profiles run while FP16 cannot initialize (Section 5.5). Third, *bandwidth-limited inference* converts the smaller weight stream into speed, and the Compact profile reaches 1.28× FP16 throughput on the L40S. Fourth, *denser provisioning* increases the number of resident operators or serving replicas under a fixed memory and power envelope. These scenarios connect the footprint result directly to query execution and infrastructure planning.

4.1. Extended workload matrix: Three datasets, two operators

To test the compression pipeline against heterogeneous vocabularies, input lengths, and structural conventions, we add a second workload family: three real-world tabular datasets crossed with two semantic operators. The two operators span the output regimes a semantic operator exhibits in practice: constrained categorical output and free-form generative output.

Datasets. **Movies** is a column of professional and viewer movie reviews from Rotten Tomatoes, and its reviews are typically one to three sentences. **Products** is product reviews from the Amazon Reviews 2023 corpus [10], sampled and prompted to elicit categorization rather than the five-word condensation used in the summarization workload. Its lengths spread wider than Movies, with occasional very long reviews acting as natural outliers. **BIRD** is question-and-answer text from the BIRD StackExchange corpus, mixing prose with inline technical content. It is the most format-heterogeneous of the three and is included to probe the limits of single-operator instance optimization (Section 7). We sample 1500 rows from each dataset for each operator.

Operators. The **Filter** operator is constrained-output: the model emits a single categorical label from a small fixed set (Positive/Negative for Movies, Positive/Negative/Neutral for Products, Yes/No for BIRD). Filter outputs are short and nearly deterministic, which makes them sensitive to distortions in the model's logit distribution. The **Projection** operator is generative: the model produces a free-text summary of the row in at most 50 words. Projection outputs are long and inherently non-deterministic, so replacement fidelity is measured against the reference model's outputs rather than a fixed gold label.

The evaluation therefore uses two complementary workload families. Table 2, the batching study, and the economic analysis use Summarization, Data Correction, and Fuzzy Join; the compression, calibration, architecture, and planner studies use the Movies/Products/BIRD × Filter/Projection matrix. We do not compare absolute throughput or fidelity across these families unless the workload and harness match.

4.2. Operator interface example

All three workloads share the same operator contract: IOLM-DB applies a fixed prompt template to a column or candidate-pair table, specializes the underlying model once, and reuses it across many invocations. The summarization workload has the following surface form. Section 6 gives the full UDF contract and execution semantics.

```
SELECT review_id,
      llm_prompt('Summarize in five words', review_text,
                 optimization_hint='speed') AS
    short_summary
FROM reviews
WHERE review_text IS NOT NULL;
```

The SQL form preserves optimizer-visible predicates and composes semantic operators with joins and aggregations in one declarative plan. Data correction and fuzzy join use the same contract with task-specific prompt templates and input columns.

The optimized operator is engine-agnostic and can also be invoked from a Python DataFrame (Section 6). A query-language host contributes optimizer visibility: predicates such as the `WHERE` clause remain in the relational plan, so filtered-out rows never issue model calls. This query-level call avoidance composes with IOLM-DB's model-level footprint reduction.

5. Evaluation

We evaluate IOLM-DB on the workloads described in Section 4. The evaluation asks whether instance-level optimization can run one intensive semantic operator at column scale while preserving reference-model behavior. We compare against per-row local inference, batch-processing strategies, and architecture-level optimization through task-specific fine-tuning. GPT-4o API calls provide teacher labels and a cloud-cost reference, not a throughput baseline in the local experiments.

5.1. Setup and measurement protocol

Metrics. Our evaluation centers on the following metrics:

- **Throughput:** rows processed per second. The short-output 8B operator is compute-bound on the A100, whereas on the L40S the reduced weight traffic raises Compact throughput by 1.28× (Section 5.5).
- **Resident model size:** the GPU-resident footprint of the operator, and the primary efficiency metric in this evaluation: a smaller operator fits more constrained hardware and leaves headroom for concurrent operators and larger key–value caches.
- **Reference fidelity:** agreement with the Llama-3.1-8B reference operator. We use exact output agreement for categorical operators and ROUGE-L for generative operators, normalized so the reference scores 1.00. Section 5.2 separately validates this replacement metric against labeled ground truth.
- **Cost efficiency:** GPU-hour cost for local inference and monetary cost for API-based inference.

Baseline comparisons. We compare IOLM-DB against three execution alternatives and one cloud reference:

Cloud Reference uses GPT-4o to generate teacher labels and estimate per-token cost under a remote API model.

Local Model Baseline deploys Llama-3.1-8B locally with standard per-row invocation.

Batch Processing Baseline groups multiple rows into a single prompt to reduce invocation overhead.

LMS Approach uses fine-tuned lightweight T5-Small students trained on teacher-generated data.

Models. We use Meta's instruction-tuned Llama-3.1-Instruct-8B model [3]. The model fits on a single modern GPU and provides a strong reference for the evaluated tasks. We verified its suitability by comparing sampled outputs with OpenAI GPT-4o and Anthropic Claude Sonnet 3.5 and manually inspecting the results.

Baseline and configurations. We execute IOLM-DB using Llama-3.1-Instruct-8B as the baseline for all metrics and evaluate two optimized variants:

- **IOLM-DB-Compact:** a 4-bit profile that minimizes resident footprint.
- **IOLM-DB-Fidelity:** an 8-bit profile that prioritizes agreement with the reference model.

These profiles expose the planner's primary footprint–fidelity choice. A mixed-precision profile provides an intermediate point (Section 5.3).

Compression search space and calibration. For each workload, we evaluate a fixed candidate set that includes quantization-only, sparsity-only, pruning-only, mixed-precision, and combined candidates. Candidate calibration follows Section 3.2.1. CBOP filters candidates by quality and memory, then minimizes predicted end-to-end cost as defined in Section 3.4.

Hardware and configuration. The primary test server contains an NVIDIA A100 80 GB GPU, two AMD EPYC 9654 CPUs, and 2 TB of main memory and runs 64-bit Debian Linux. The software stack comprises Hugging Face Transformers 4.57.6, compressed-tensors 0.14, CUDA 12.8 (driver branch 595), and vLLM 0.19.0. The *Emulated Memory Budgets* experiment caps the A100 memory pool at 24, 32, and 48 GB to isolate capacity effects. The *Hardware-Dependent Throughput and Kernel Compatibility* experiment then repeats the key workload on an NVIDIA L40S, varying both architecture and memory bandwidth, and verifies the current kernel boundary on a V100.

Table 1
Generation-time decomposition (Eq. (5)) for representative recipes, in seconds per phase. The *total* is measured from the run logs; the per-phase split is apportioned by the pipeline fractions the current logs support, since the compression pass and calibration/serialization phases are not separately timestamped in this prototype. $T_{\text{candidate}}$ (the quantization pass or student training) dominates at ≈86% of the total, while T_{sample} and $T_{\text{serialize}}$ are small. Measured totals range from 1.1 min for LMS to 25.7 min for W8A8.

Recipe	T_{sample}	$T_{\text{candidate}}$	T_{validate}	$T_{\text{serialize}}$	Total (min)
8-bit GPTQ (IOLM-DB-Fidelity)	42	902	63	42	17.5
4-bit GPTQ (IOLM-DB-Compact)	39	844	59	39	16.3
LMS (T5-Small distill)	3	59	4	3	1.1
W8A8 (activation quantization)	62	1,326	92	62	25.7

Harness and profile provenance. The controlled compression ablation (Table 4) is the canonical 1500-row, 128-calibration-row comparison for the extended workload matrix. Calibration-source, precision-scheme, engine-integration, and memory-cap experiments use separate controlled harnesses to answer different questions; their fidelity values are therefore not pooled into a single estimate. Captions identify the applicable row count, scheduler, calibration size, or host engine. CBOP compares throughput only within a common harness and stores the source of each profiled quantity; Table 15 makes that provenance explicit where a hardware throughput profile is paired with an independently measured fidelity profile for the same strategy and operator.

One-time generation cost. Table 1 makes CBOP's one-time generation cost explicit. In the representative generation-cost harness, the weight-only quantized candidates take 16–18 min to produce, dominated by the GPTQ pass, while the LMS candidate takes 1.1 min. Controlled harnesses that also vary calibration size or precision scheme have their own logged generation times, reconciled below. CBOP combines the profile-matched cost with measured throughput and memory fit: the high-fidelity A100 profiles select compression for capacity, and the L40S profile yields a finite throughput break-even of approximately 554,000 rows for the 4-bit operator. Section 5.3 reports the separate W8A8 throughput-positive point.

The unbounded A100 break-even has a direct planning interpretation. For a compressed candidate whose measured r_s does not exceed r_b, the denominator in Eq. (3) is nonpositive, so no row count can recover G_s through execution time alone. CBOP therefore excludes that candidate from a latency-minimizing A100 plan while retaining it when the 14.98 GB baseline violates the memory constraint or prevents co-residency. On the L40S, $r_s > r_b$, the denominator becomes positive, and the same equation returns a usable cardinality threshold. The planner consequently requires no compression-specific fallback rule: measured throughput, fidelity, and memory feasibility determine the choice in both regimes.

Worked predictions for the throughput-positive strategies. On Movies/Projection in PySpark, batching raises the FP16 baseline from 89.6 to 91.4 rows/s with $G_s = 0$, so CBOP selects it immediately. On the L40S, 4-bit compression raises throughput from 124.8 to 160.1 rows/s, and with $G_s = 16.3$ min Eq. (3) gives $N^* \approx 554,000$ rows. On Products/Projection, T5-Small is rejected because both its throughput and fidelity trail the teacher, while Qwen2.5-0.5B is rejected by the quality floor. These cases exercise each stage of the decision rule: generation-cost amortization, memory feasibility, throughput, and fidelity.

Generation time is profile- and harness-specific rather than a property of bitwidth alone. For Compact, the complete set of reported totals is 15.4–16.4 min in the per-workload ablation, the representative 16.3 min recipe used in the L40S break-even calculation, 18.9 min in the separately instrumented 128-row calibration-size sweep, and 22.3 min in the precision-scheme harness. That last harness reports Mixed at 22.9 min under the same instrumentation, enabling the direct Compact–Mixed comparison in Section 5.3. CBOP stores and uses the generation time from the matching strategy profile, so these controlled measurements are not substituted across harnesses.

5.2. Footprint, fidelity, and task quality

Table 2 reports GPU-resident model size, reference fidelity, and throughput for the baseline, compressed, and distilled model variants.

Table 2 shows a consistent 1.8–2.8× footprint reduction. The Fidelity profile preserves 0.94–0.99 of reference behavior at 8.46 GB. Compact reduces the footprint to 5.31 GB and is strongest on categorical Filter operators, reaching 0.96–0.98 on Movies and Products Filter, while generative Projection workloads favor Fidelity or the mixed-precision profile. Fuzzy join is the exception among closed-label operators at 4-bit (0.83): unlike the Filter operators, whose input is a single column value and whose output is one label token, its input is a concatenation of two entity descriptions, so both the input length and the decision boundary differ from the single-column case. We report the gap rather than attribute it to one cause, and we therefore scope the 4-bit categorical claim to the Filter operators throughout. A controlled pair-format and input-length ablation remains future work. CBOP encodes this separation directly through its quality and memory constraints.

Section 5.5 next shows how the footprint and fidelity measurements in Table 2 translate into feasibility and throughput on lower-bandwidth hardware; the native 3B alternatives are reported separately in Table 7.

Validation against labeled ground truth. The primary quality metric is *fidelity to the uncompressed Llama-3.1-8B reference*, normalized so the reference scores 1.00. This metric directly measures whether a compressed artifact can replace the reference operator. For the Movies and Products Filter workloads, sentiment labels also permit an independent task-quality measurement. Table 3 reports both quantities for the reference and compressed profiles.

The ground-truth measurements validate reference fidelity as a task-quality proxy on the labeled Filter workloads. The reference model reaches 0.815 absolute accuracy on Movies and 0.883 on Products. Fidelity reaches 0.816 and 0.883, and Compact reaches 0.809 and 0.877. Both compressed profiles therefore remain within 0.006 of the reference model's absolute accuracy. The 8-bit profile even slightly exceeds the reference on Movies, showing that agreement with the reference is not systematically optimistic about task quality. For generative Projection, reference agreement measures replacement fidelity, and human evaluation would separately tell us which paraphrases readers accept.

5.3. Where the compression gain comes from

The results above report the deployed profiles. This subsection separates the three compression families that produce them, measures what column calibration adds over generic calibration at equal bitwidth, and reports the intermediate precision points the planner can serve. The calibration and bitwidth-selection procedures themselves are defined once in Section 3.

To isolate each technique on the extended workload matrix (Section 4.1), and to expose the per-technique generation cost and fidelity that determine when optimization amortizes, we run a controlled ablation on Llama-3.1-Instruct-8B across the Movies and Products datasets and both the Filter and Projection operators. Each row of Table 4 is one isolated configuration: 8-bit GPTQ only, 4-bit GPTQ only, 50% SparseGPT unstructured sparsification only, 20% LLM-Pruner structural pruning only, and the combined IOLM-DB-Compact selection pipeline. The combined pipeline evaluates the three compression families in sequence but serializes only stages that pass artifact and serving validation; in the current stack, only 4-bit quantization survives. We report generation time (the one-time cost of producing the operator), end-to-end throughput, and fidelity normalized to the uncompressed reference.

Table 2
GPU-resident footprint, reference fidelity, and throughput (rows/s) for the local reference, compressed IOLM-DB profiles, and LMS students. Compact is the 4-bit minimum-footprint profile; Fidelity is the 8-bit fidelity-oriented profile. Batch-only configurations appear in Table 17, and monetary cost appears in Table 13.

Workload	Model variant	Model size	Ref. fidelity	Throughput
Summarization	Baseline	14.98 GB	1.00	120.4
	IOLM-DB-Compact (4-bit)	5.31 GB	0.73	88
	IOLM-DB-Fidelity (8-bit)	8.46 GB	0.97	91
	LMS (T5-Small)	0.23 GB	0.24	139
Data Correction	Baseline	14.98 GB	1.00	169.9
	IOLM-DB-Compact (4-bit)	5.31 GB	0.59	126
	IOLM-DB-Fidelity (8-bit)	8.46 GB	0.94	120
	LMS (T5-Small)	0.23 GB	0.12	71
Fuzzy Join	Baseline	14.98 GB	1.00	748.7
	IOLM-DB-Compact (4-bit)	5.31 GB	0.83	464
	IOLM-DB-Fidelity (8-bit)	8.46 GB	0.99	522
	LMS (T5-Small)	0.23 GB	0.91	1007

Table 3
Reference fidelity versus absolute accuracy against ground-truth labels on the two labeled Filter workloads (1500 rows each). The compressed operators remain within 0.006 of the FP16 reference's absolute accuracy, validating reference fidelity as a task-quality proxy for these workloads.

Workload	Variant	Agreement w/ reference	Absolute acc. (ground truth)
Movies/Filter	Reference (FP16)	1.00	0.815
	IOLM-DB-Fidelity (8-bit)	0.995	0.816
	IOLM-DB-Compact (4-bit)	0.968	0.809
Products/Filter	Reference (FP16)	1.00	0.883
	IOLM-DB-Fidelity (8-bit)	0.997	0.883
	IOLM-DB-Compact (4-bit)	0.983	0.877

Table 4
Per-technique compression ablation on Llama-3.1-Instruct-8B over Movies and Products for the Filter and Projection operators. Each row is one isolated compression configuration. Generation time is the one-time cost of producing the compressed operator; throughput is end-to-end rows per second on the A100; fidelity is normalized to the uncompressed reference (1.00). Each cell is measured over 1500 rows with a 128-row calibration set, 64 output tokens, and the vLLM 0.19 default scheduler (no explicit sequence or batched-token cap).

Dataset	Operator	Configuration	Gen. Time (min)	Throughput (rows/s)	Ref. Fidelity
Movies	Filter	Baseline (uncompressed)	–	289.4	1.00
		Quant 8-bit (GPTQ)	14.9	264.8	0.99
		Quant 4-bit (GPTQ)	15.5	235.8	0.96
		Sparsify 50% (SparseGPT)	3.3	101.3	0.01
		Prune 20% (LLM-Pruner)	1.6	274.4	1.00
		Combined (IOLM-DB-Compact)	15.5	231.7	0.97
	Projection	Baseline (uncompressed)	–	141.9	1.00
		Quant 8-bit (GPTQ)	15.0	127.3	0.93
		Quant 4-bit (GPTQ)	15.4	121.0	0.61
		Sparsify 50% (SparseGPT)	3.0	101.2	0.13
		Prune 20% (LLM-Pruner)	1.6	137.2	0.99
		Combined (IOLM-DB-Compact)	15.4	122.1	0.63
Products	Filter	Baseline (uncompressed)	–	278.9	1.00
		Quant 8-bit (GPTQ)	17.5	254.5	1.00
		Quant 4-bit (GPTQ)	15.7	220.6	0.98
		Sparsify 50% (SparseGPT)	3.4	98.7	0.00
		Prune 20% (LLM-Pruner)	1.6	264.4	1.00
		Combined (IOLM-DB-Compact)	15.3	219.2	0.98
	Projection	Baseline (uncompressed)	–	149.1	1.00
		Quant 8-bit (GPTQ)	14.8	135.0	0.96
		Quant 4-bit (GPTQ)	15.6	128.3	0.67
		Sparsify 50% (SparseGPT)	3.4	91.5	0.15
		Prune 20% (LLM-Pruner)	1.6	145.5	0.99
		Combined (IOLM-DB-Compact)	16.4	131.5	0.68

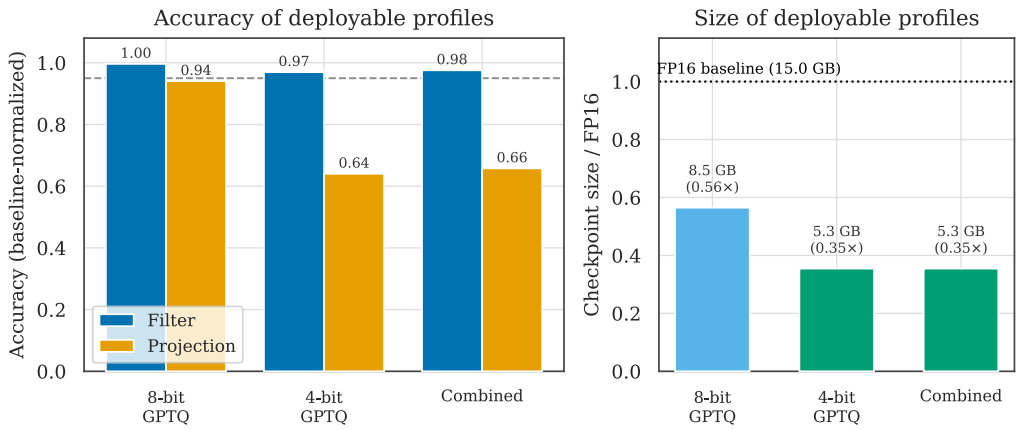


Fig. 2. Deployable quantized profiles from the Llama-3.1-8B compression ablation (fidelity averaged over Movies and Products). **Left:** reference fidelity for Filter and Projection. **Right:** serialized checkpoint size relative to the 14.98 GB FP16 checkpoint. The figure focuses on the candidates that produce compact, servable artifacts: 8-bit GPTQ, 4-bit GPTQ, and the selected Combined profile. Sparse and pruned candidates are retained in Table 4 as negative ablation results but omitted here because they do not reduce the deployed artifact. The Combined profile coincides with 4-bit GPTQ in the current stack.

Three findings stand out. Fig. 2 visualizes the deployable quantized profiles, while Table 4 retains all negative ablation results. First, *the tolerable precision depends on the operator's output regime*. 8-bit quantization is near-lossless on both operators (0.99–1.00 on Filter, 0.93–0.96 on Projection). Dropping to 4-bit leaves the categorical Filter operator almost unaffected (0.96–0.98) but sharply reduces agreement on generative Projection (0.61–0.67): a constrained label decision tolerates logit perturbation better than autoregressive free-text generation. This gap motivates separate Fidelity and Compact profiles.

Second, *the structure-level techniques do not improve the deployed artifact*. Structural pruning is inexpensive to apply (about 1.6 min) and preserves 0.99–1.00 fidelity, but the served checkpoint remains approximately 15.0 GB and throughput does not increase. At 50% unstructured sparsity, the runtime falls back to an unaccelerated path: artifact size grows by 93%, throughput declines, and fidelity falls to 0.00–0.15. These measurements remove both techniques from the current production recipe. Hardware-supported 2:4 sparsity is a future candidate and is not part of the evaluated search space.

Third, *the combined IOLM-DB-Compact operator is, empirically, the 4-bit GPTQ operator*. Its resident size (5.3 GB) equals the 4-bit-only configuration's, and its fidelity lies within ± 0.02 of it in every cell. Because the sparsification and pruning stages do not survive into a servable artifact, the production recipe is 4-bit quantization with IOLM-DB's column-aware calibration and planning. The separate combined row documents that selection outcome.

No compressed configuration in Table 4 exceeds the continuous-batched FP16 baseline on the A100. The W4A16 GPTQ kernel stores weights in 4 bits but performs the matrix multiplication in FP16, dequantizing weights into registers and applying per-group scale and zero-point factors at every layer. This design accelerates execution only when reduced weight traffic repays the dequantization overhead. The A100 run is compute-efficient and has ample memory capacity: its FP16 path uses highly optimized cuBLAS and FlashAttention-2 kernels, and the full model leaves sufficient room for the active key–value cache. The smaller weight stream therefore does not increase the effective batch, while dequantization remains on the critical path. The ablation establishes a $2.8\times$ capacity gain on this hardware, and Section 5.5 shows the complementary bandwidth-limited regime in which the same 4-bit artifact also raises throughput.

Incremental benefit over generic GPTQ. Because the combined recipe reduces to 4-bit GPTQ (finding three above), a fair question is what IOLM-DB adds over running off-the-shelf GPTQ with a standard, task-agnostic calibration set. We answer it with a controlled three-arm comparison at a fixed 4-bit bitwidth and serving kernel, so only the calibration set varies, and we repeat every cell over three independent calibration draws (seeds) to separate a real effect from GPTQ's run-to-run variance. **GPTQ-generic** calibrates on a general-purpose text corpus (128 sequences sampled across WikiText-2), the conventional choice in the quantization literature. **GPTQ-column** (IOLM-DB) calibrates on a 128-row sample drawn from the target column itself (Section 3.2.1). **GPTQ-column-IO** additionally appends the reference model's output on each calibration row, so the calibration sequences carry the operator's decode-phase distribution and not only its prompt (Section 3.2.1). Holding everything else fixed isolates the contribution of column calibration, which is the instance-optimization step this paper actually introduces.

This comparison quantifies the contribution beyond a GPTQ-only baseline. Column calibration improves mean reference fidelity at equal bitwidth and footprint in all six cells, and the six same-signed outcomes yield a one-sided sign-test result of $p = 0.016$; the mean gain is +0.036. This across-cell sign test does not imply that every individual cell is statistically separated with three draws. Two structural patterns distinguish the result from a uniform offset. First, the largest benefit occurs where generic calibration is least reliable: on Movies/Filter, the three WikiText draws span 0.80–0.93 (standard deviation 0.071), whereas the column draws span 0.956–0.969 (standard deviation 0.006). Column calibration therefore improves both mean fidelity and reproducibility for a fixed prompt–column pair. Second, the output-aware variant isolates a generation-specific mechanism. GPTQ estimates layer-wise quantization error from forward passes over calibration sequences that normally end with the prompt, omitting the model's decode-phase activations. Appending the reference completion supplies those activations. Consistent with this mechanism, output-aware calibration improves every Projection cell by +0.009 to +0.053 over column-only calibration and changes the single-token Filter cells by at most 0.007. Column-plus-output calibration is therefore the strongest evaluated 4-bit generative recipe at unchanged bitwidth and footprint. Together, the source and output-aware results establish calibration-set construction as the measured instance-optimization contribution over column-agnostic GPTQ.

Calibration-set-size sensitivity. Every result above uses a 128-row calibration set, following the convention in the GPTQ literature. Sweeping Products/Projection over 128, 256, 512, and 1024 rows at a fixed GPTQ recipe and seed changes agreement by at most 0.022 while tripling generation time (Table 18). The 128-row set is therefore the efficient operating point for this workload. Higher fidelity requires more precision, as provided by the mixed and Fidelity profiles, rather than a larger calibration budget.

Table 5
Incremental benefit of column calibration over generic calibration at fixed 4-bit GPTQ, as mean ± standard deviation over three independent calibration draws. All arms use identical bitwidth and serving kernel; only the calibration set differs. Fidelity is normalized to the uncompressed Llama-3.1-8B reference. Column calibration is higher than generic in all six cells (one-sided sign test $p = 0.016$); the output-aware variant adds a further gain on every generative Projection cell (bold) while leaving the constrained Filter operator unchanged. Measured over 1500 evaluation rows with a 128-row calibration set.

Dataset	Operator	GPTQ-generic (WikiText)	GPTQ-column (column)	Δ (col−gen)	GPTQ-column-IO (column + output)
Movies	Filter	0.885 ± 0.071	0.962 ± 0.006	+0.076	0.968 ± 0.005
	Projection	0.601 ± 0.011	0.635 ± 0.011	+0.034	**0.644** ± 0.014
Products	Filter	0.979 ± 0.003	0.986 ± 0.001	+0.007	0.986 ± 0.002
	Projection	0.652 ± 0.006	0.665 ± 0.038	+0.013	**0.718** ± 0.016
BIRD	Filter	0.884 ± 0.025	0.914 ± 0.003	+0.030	0.912 ± 0.009
	Projection	0.561 ± 0.013	0.614 ± 0.010	+0.053	**0.632** ± 0.003

Per-layer mixed precision. Section 3.2.1 defines the per-layer sensitivity map. Here we serve one footprint-efficient mixed-precision point. The recipe keeps the relatively small attention projections at 8-bit and stores the parameter-heavy feed-forward matrices at 4-bit. Although W_{down} and W_{up} are highly sensitive, protecting these largest matrices would approach the fully 8-bit footprint. Assigning 8-bit precision to the attention block instead adds precision at lower storage cost. At 5.92 GB, the resulting profile improves Projection fidelity over uniform 4-bit on all three datasets, Products 0.665 → **0.747**, Movies 0.635 → 0.664, and BIRD 0.614 → 0.650, while remaining near-lossless on Filter (0.93–0.99). On Products/Projection it runs at 136.5 rows/s with a 22.9-minute generation time, compared with 132.7 rows/s and 22.3 min for Compact in the same precision-scheme harness. Consequently, at $N = 10^6$ and a 6.0 GB budget, CBOP selects Mixed whenever both candidates meet the fidelity floor; the choice follows measured end-to-end time rather than an unstated tie-break.

Activation-quantized throughput point. The candidate search also evaluates W8A8 quantization on Products/Projection. On the same 1500-row, 64-output-token vLLM 0.19 default-scheduler harness used for this sweep, it reaches 206.1 rows/s on the A100, compared with 146.3 rows/s for FP16, at 0.785 reference fidelity and an 8.46 GB footprint. This 1.41× result is the fastest compressed A100 operating point, but its fidelity excludes it from the default $\alpha_{\min} = 0.90$ regime. With a 0.75 floor and one million rows, CBOP admits W8A8 and selects it after its approximately 777,000-row throughput break-even, including the measured 25.7-minute generation cost. The result demonstrates why the planner jointly evaluates throughput and fidelity rather than treating compression as a binary memory decision.

5.4. Generality: Architectures, small models, and distillation

The results so far fix the architecture to Llama-3.1-8B and the strategy to compression. This subsection tests whether the pattern holds for other reference architectures, how the compressed operator compares with models that are natively small, and where distillation is a viable alternative.

Other reference architectures. The ablation above fixes the architecture to Llama-3.1-8B. To test architectural generality, we build the same column-calibrated 4-bit Compact profile for Qwen2.5-7B and Gemma-2-9B across Movies, Products, and BIRD and both operators. Fidelity is normalized to each architecture's own uncompressed reference. Table 6 reports the results.

The Llama-3.1-8B pattern generalizes across architectures. Filter fidelity remains 0.97–0.99 for Qwen2.5-7B and 0.95–0.99 for Gemma-2-9B, confirming that categorical operators tolerate the 4-bit profile. Projection remains more precision-sensitive (0.60–0.69 for Qwen2.5-7B and 0.69–0.81 for Gemma-2-9B), with Gemma-2-9B retaining the highest generative fidelity. The resulting footprints are 5.16 GB and 5.71 GB, respectively, establishing a consistent 2–3× reduction across all three architectures.

What BIRD reveals about the limits of single-operator calibration. BIRD combines prose, inline code, equations, and markup and has the longest, most variable rows in the matrix. It produces the lowest throughput for both architectures: 99 rows/s for Qwen2.5-7B and 62 rows/s for Gemma-2-9B. Its Projection fidelity is also architecture-sensitive. Qwen reaches 0.62 on BIRD, between Movies (0.60) and Products (0.69), while Gemma reaches 0.69, below Movies (0.77) and Products (0.81). In contrast, BIRD Filter remains high at 0.95–0.97. These results locate the boundary more precisely. Heterogeneous formatting consistently increases execution cost, constrained categorical outputs hold up across both architectures, and open-ended generation depends more strongly on the base architecture. BIRD therefore motivates the per-cluster calibration strategy described in Section 7 without attributing every fidelity difference to heterogeneity alone.

Compressed LLM vs. natively small models. A natural alternative to compressing an 8B model is to begin with a smaller model. We therefore compare Compact (4-bit Llama-3.1-8B, 5.31 GB) with two uncompressed, instruction-tuned 3B models, Qwen2.5-3B and Llama-3.2-3B, on Movies and Products. All three are scored against the uncompressed Llama-3.1-8B reference. Table 7 reports fidelity and throughput averaged across the two datasets.

The comparison splits by operator type. Native 3B models reach approximately 600–650 rows/s on Filter, compared with approximately 225 rows/s for Compact, but retain only 0.27–0.42 fidelity on Projection, compared with 0.66 for the compressed 8B. On Filter, the gap narrows to 0.89 versus 0.98. Compact is also smaller than the FP16 native models (5.31 GB versus 6.2–6.4 GB). CBOP therefore routes throughput-oriented Filter operators to the native 3B and fidelity-oriented Projection operators to the compressed 8B.

Batch size. Batching depends on workload shape, and the best batch size is workload-dependent: summarization peaks at batch size 20 in these measurements, data correction continues to improve through batch size 50, and fuzzy join peaks at 20 and then loses throughput at 50 (Table 17). These results support batching as a useful local optimization. Reducing the resident model footprint addresses a different bottleneck.

Lightweight model synthesis (LMS). LMS complements model compression by changing the architecture rather than only compressing the reference model. Building on ScaleLLM [20], it automates distillation for database workloads. With T5-Small as the base architecture, LMS reduces model size while retaining task-relevant behavior.

LMS trains from teacher labels generated on representative samples, avoiding manual annotation. The distilled students reduce resident footprint to 0.23 GB for T5-Small and 0.92 GB for Qwen2.5-0.5B, 65× and 16× smaller than the 8B reference, respectively. Table 8 reports the Products/Projection experiment, where fidelity is normalized to the uncompressed Llama-3.1-8B teacher.

T5-Small reaches 0.37 fidelity on Products/Projection from 100 training examples, compared with 0.66 for the compressed 8B, and

Table 6
Architectural generalization of the column-calibrated 4-bit Compact profile. Size is resident footprint; fidelity is normalized to each architecture's own uncompressed reference; throughput is rows/s on the A100.

Architecture	Dataset	Operator	Size (GB)	Ref. Fidelity	Throughput (rows/s)
Qwen2.5-7B	Movies	Filter	5.16	0.97	256
		Projection	5.16	0.60	169
	Products	Filter	5.16	0.99	234
		Projection	5.16	0.69	182
	BIRD	Filter	5.16	0.97	149
		Projection	5.16	0.62	99
Gemma-2-9B	Movies	Filter	5.71	0.98	186
		Projection	5.71	0.77	119
	Products	Filter	5.71	0.99	170
		Projection	5.71	0.81	121
	BIRD	Filter	5.71	0.95	104
		Projection	5.71	0.69	62

Table 7
Compressed Llama-3.1-8B operator versus natively small uncompressed models. Fidelity is normalized to the uncompressed Llama-3.1-8B reference and averaged over Movies and Products. Native-model sizes are approximate safetensors weights in FP16; Compact size is the 4-bit artifact footprint.

Model	Size (GB)	Operator	Ref. Fidelity	Throughput (rows/s)
Qwen2.5-3B (uncompressed)	6.2	Filter	0.89	652
		Projection	0.27	396
Llama-3.2-3B (uncompressed)	6.4	Filter	0.89	604
		Projection	0.42	264
IOLM-DB-Compact (4-bit Llama-3.1-8B)	5.31	Filter	0.98	225
		Projection	0.66	127

Table 8
LMS on Products/Projection with 100 teacher-labeled examples and a held-out evaluation split. Fidelity is normalized to the Llama-3.1-8B teacher; generation time combines teacher labeling and student training. The Qwen row records the failed FP16 training configuration analyzed below.

Student	Size (GB)	Ref. Fidelity	Throughput (rows/s)	Gen. Time (min)
T5-Small	0.23	0.37	99.8	1.1
Qwen2.5-0.5B (failed FP16 run)	0.92	0.00 (failed)	327.2	1.1

runs at 99.8 rows/s compared with approximately 149 rows/s for the teacher. The recorded Qwen2.5-0.5B run collapses to 0.00 fidelity despite reaching 327.2 rows/s. Inspection identifies unstable full-parameter FP16 AdamW training: the student emits degenerate outputs after the update sequence. This row records a failed training configuration rather than a capacity result. No working Qwen student was produced in this evaluation; the released training path has been corrected to use BF16 or FP32 master weights, but its outcome is not reported here. Accordingly, the reported LMS evidence establishes T5-Small as a viable strategy for categorical and closed-label operators; it does not support generalization to free-form generation at the evaluated supervision budget.

The throughput comparison is implementation-level, not a claim that a 60M-parameter network intrinsically requires more computation than an 8B network. The T5 student uses Hugging Face `generate` with fixed batches of 32, whereas the Llama reference uses vLLM continuous batching; the smaller student can therefore trail the teacher when its serving path is less optimized. CBOP profiles the realized implementation, so it does not infer throughput from parameter count.

LMS fidelity by output regime. The reliable LMS result is the contrast between constrained and generative outputs. T5-Small reaches 0.91 on Fuzzy Join but only 0.24 on Summarization, 0.12 on Data Correction, and 0.37 on Products/Projection. On Fuzzy Join, the 0.23 GB student also exceeds Compact's 0.83 fidelity, making distillation the stronger measured option when this closed-label decision boundary is covered by the teacher-labeled sample. Three mechanisms are consistent with this pattern. First, 100 teacher-labeled examples densely cover a two- or three-label output space but sparsely cover an open set of summaries and corrections. Second, the 60M-parameter student is learning an 8B teacher's complete generation behavior rather than a compact decision boundary. Third, ROUGE-L penalizes valid paraphrases, although scores of 0.12–0.37 remain too low for a fidelity-sensitive replacement even under that metric. These results define LMS as a production option for categorical Filter and matching operators at the evaluated budget. Generative operators instead use the Fidelity or mixed-precision profile. This placement makes LMS a conditional architecture-level strategy within CBOP, rather than a universal stage of the compression pipeline.

Cross-strategy takeaways. No strategy dominates across operator and hardware regimes. Fidelity provides the strongest general replacement at 8.46 GB. Compact minimizes footprint at 5.31 GB and is the preferred compressed profile for categorical operators. The mixed profile improves all three generative Projection cells over uniform 4-bit at 5.92 GB. W8A8 supplies the fastest A100 Projection point when a 0.785 fidelity level is acceptable. Native 3B models maximize Filter throughput, and LMS reaches the smallest footprint for constrained output spaces. Hardware completes the decision: batching wins on the unconstrained A100 at high fidelity, whereas Compact delivers both feasibility and 1.28× throughput on the lower-bandwidth L40S. CBOP selects among these measured operating points rather than assigning one optimization to every workload.

5.5. Deployment: Memory, hardware, engines, and economics

The preceding subsections measure the operator in isolation on one accelerator. This subsection asks what the smaller artifact changes in

Table 9
Products/Projection throughput (rows/s) over 500 rows on an NVIDIA A100 with vLLM's GPU memory pool capped to the stated budget. Every cap uses the same vLLM 0.19 configuration: 64 output tokens, `max_model_len`=4096, `max_num_seqs`=64, and `max_num_batched_tokens`=8192. Holding the processor and harness fixed isolates the capacity effect within this sweep: the 5.31 GB Compact operator tracks the 14.98 GB FP16 reference at every budget, with reference fidelity of 0.69.

Emulated memory budget	Baseline (14.98 GB)	IOLM-DB-Compact (5.31 GB)	Speedup
24 GB	107.3	108.1	1.01×
32 GB	108.0	109.3	1.01×
48 GB	108.0	108.8	1.01×

deployment: which memory budgets it runs under, how it behaves on a second GPU architecture, whether it drops into existing engines unchanged, and what it costs.

Emulated memory budgets. We first isolate memory capacity from GPU architecture by running Products/Projection on the A100 with vLLM's memory pool capped at 24, 32, and 48 GB. Both the 14.98 GB FP16 operator and the 5.31 GB Compact operator fit at these budgets, which permits a controlled throughput comparison while holding compute capability, memory bandwidth, and software constant. The 16 GB class is evaluated separately in the hardware-dependent experiment reported in Table 10, because FP16 leaves insufficient memory for the CUDA context, captured graphs, and key–value cache at that budget.

Table 9 reports a stable 1.01× Compact/FP16 ratio at all three caps. The absolute 108.0 rows/s baseline should not be compared with the 149.1 and 146.3 rows/s baselines above: those runs use 1500 rows and the default vLLM scheduler, whereas the cap sweep uses 500 rows and the explicit bounds in the caption. The controlled claim is the within-sweep result: changing only the pool cap from 24 to 48 GB does not change A100 throughput. Products/Projection emits at most 50 words, so prefill compute dominates and the active key–value cache remains small. Even the 24 GB cap leaves enough cache capacity for the offered concurrency, so releasing 9.7 GB does not increase the effective batch. This result separates memory *capacity* from memory *bandwidth*: Table 10 changes the GPU and shows that the reduced weight stream produces a throughput gain when bandwidth becomes more constraining.

The constrained-memory results translate into four deployment benefits:

- **Hardware class.** Under the measured 14.8 GB cap in Table 10, both compressed profiles run and FP16 cannot initialize.
- **Operator co-residency.** Three distinct Compact operators occupy 15.93 GB, compared with 44.87 GB for three FP16 copies.
- **Provisioning density.** The same memory budget can host more specialized operators or inference replicas, increasing cluster-level concurrency.
- **Cache headroom.** The released memory can support larger key–value caches and batches for workloads whose concurrency or output length makes cache capacity binding.

Hardware-dependent throughput and kernel compatibility. The A100 cap experiment holds the processor constant. We next run the same 500-row Products/Projection workload on an NVIDIA L40S, whose Ada architecture belongs to the same compute-capability class as the L4 and RTX 40 series and whose memory bandwidth is less than half the A100's. We cap the available pool at 22.2 and 14.8 GB to represent 24 and 16 GB deployment classes. A V100 run tests the stated compute-capability boundary directly. Table 10 reports the resulting hardware-specific profiles.

At 14.8 GB, vLLM reports negative available key–value-cache memory for FP16 after allocating weights, the CUDA context, and captured graphs, and the engine refuses to start. Compact and Fidelity both initialize and execute at the same cap. This measurement establishes a hard feasibility boundary: compression changes the deployment from unsupported to serviceable within the 16 GB memory class.

At 22.2 GB, where all three profiles fit, Compact processes 160.1 rows/s compared with 124.8 rows/s for FP16, a 1.28× gain. The same ratio persists at the lower cap, showing that the gain follows the processor rather than capacity alone. The L40S provides approximately 860 GB/s of memory bandwidth compared with about 1900 GB/s on the A100. Compact reduces weight traffic by 65%, enough to repay W4A16 dequantization in this regime. Fidelity reduces weight traffic by 44% but reaches 118.5 rows/s, slightly below FP16, which supplies a useful control for the bandwidth explanation.

The measured L40S rates also give CBOP a finite throughput break-even. With $G_s = 16.3$ min, $r_b = 124.8$ rows/s, and $r_s = 160.1$ rows/s, Eq. (3) yields $N^* \approx 554{,}000$ rows. Below that cardinality, Compact remains a capacity option. Above it, Compact wins on both capacity and predicted execution time.

Kernel compatibility defines a separate boundary. The compressed-tensors W4A16 and W8A16 kernels require compute capability ≥ 7.5. The V100 (cc 7.0) serves FP16 at 52.0 rows/s but rejects both compressed checkpoints, whereas Turing and newer NVIDIA families meet the requirement. This is an implementation constraint of the current artifact and is included explicitly in CBOP's hardware-feasibility test.

Why the L40S is the right proxy for a consumer card. The L40S is a data-center product, so we state precisely what it does and does not establish about consumer hardware. It shares the compatibility property that determines whether the current serving library admits the compressed artifact: compute capability 8.9, the same Ada generation as the L4 and RTX 40 series. The memory budgets we impose, 22.2 and 14.8 GB, bracket the 24 and 16 GB classes those cards ship in. The measured footprints establish the capacity advantage, but an untested card may select a different optimized kernel and therefore requires its own serving validation. On the tested L40S, the engine confirms the boundary directly: a 5.31 GB operator runs where the 14.98 GB operator cannot allocate a key–value cache.

The 1.28× throughput ratio is the claim that does not transfer unchanged. At approximately 860 GB/s the L40S sits within the same bandwidth band as the consumer flagships, which run between roughly 930 and 1010 GB/s, but each card pairs that bandwidth with different compute throughput, and the gain from a smaller weight stream depends on the ratio between the two rather than on bandwidth alone. We therefore report 1.28× as a measurement on one Ada card in a bandwidth-limited regime, and we do not claim a specific speedup for any consumer GPU. Cards with a higher bandwidth-to-compute ratio than the L40S should be expected to gain less, and CPU-only hosts remain outside the evaluated space entirely.

Integration in a DataFrame engine. To show that an instance-optimized operator drops into a real execution engine, we run IOLM-DB inside PySpark: Spark performs the relational scan, filter, and reorder steps, while the compressed operator is invoked in-process, one resident checkpoint at a time. Table 11 reports five strategy rows on the Movies dataset (15,000 rows, Projection operator), including the planner's selected row.

The PySpark sweep reproduces the A100 pattern inside a DataFrame engine. Compact and Fidelity run at 0.84× and 0.96× baseline throughput, respectively, while batching reaches 1.02× at 0.99 fidelity. With 80 GB available and a quality-sensitive Projection query, CBOP selects batching, exactly matching the measured optimum. Under a binding memory budget, the same planner instead selects Fidelity or Compact according to the requested fidelity floor, as evaluated by the CBOP decision procedure in Section 5.6.

Compact records 0.49 fidelity on the full 15,000-row integration set, compared with 0.63 on the 1500-row ablation sample. The larger set contains more long and atypical reviews and provides the more

Table 10
Products/Projection throughput (rows/s; 500 rows per cell) across two hardware architectures and constrained memory pools. On the L40S, Compact is 1.28×faster at 22.2 GB and remains serviceable at 14.8 GB, where FP16 cannot initialize. The V100 directly confirms that the current compressed-tensors kernels exclude compute capability 7.0.

GPU (memory)	Baseline FP16 (14.98 GB)	IOLM-DB-Compact 4-bit (5.31 GB)	IOLM-DB-Fidelity 8-bit (8.46 GB)	Notes
L40S (cc 8.9), 22.2 GB cap	124.8	**160.1**	118.5	all three fit
L40S (cc 8.9), 14.8 GB cap	**cannot serve**	159.0	115.1	FP16 key–value cache < 0
V100 (cc 7.0), 28.6 GB cap	52.0	incompatible	incompatible	kernel requires cc ≥ 7.5

Table 11
IOLM-DB strategies executed inside PySpark on Movies (15,000 rows, Projection). Spark performs the relational steps, and a resident vLLM engine executes one checkpoint at a time. Reference fidelity is ROUGE-L against the uncompressed operator; speedup is relative to that operator.

Strategy	Size (GB)	Ref. Fidelity	Throughput (rows/s)	Speedup
Baseline (Llama-3.1-8B)	14.98	1.00	89.6	1.00×
IOLM-DB-Compact (4-bit)	5.31	0.49	75.6	0.84×
IOLM-DB-Fidelity (8-bit)	8.46	0.93	85.9	0.96×
Batch (uncompressed)	14.98	0.99	91.4	1.02×
CBOP (planner choice)	14.98	0.99	91.4	1.02×

Table 12
The same FlockMTL semantic query over 1500 Products rows with three interchangeable vLLM backends. Fidelity is measured against FlockMTL's FP16 run, isolating the checkpoint substitution from host-level prompt templating. The compressed backends reduce resident footprint by up to 2.8×while preserving 0.979 fidelity on the constrained Filter operator. FlockMTL's serialized scalar-request timing is excluded because it is not comparable to the continuous-batching throughput experiments.

Backend (inside FlockMTL)	Footprint (GB)	Filter fidelity	Projection fidelity
FP16 Llama-3.1-8B	14.98	1.00	1.00
IOLM-DB-Fidelity (8-bit)	8.46	0.997	0.897
IOLM-DB-Compact (4-bit)	5.31	0.979	0.638

stringent estimate for this deployment, reinforcing the use of Fidelity for quality-sensitive generation. This measured host-engine substitution shows that the operator interface is portable while the planner remains responsive to the deployment's observed quality profile.

Composability: A compressed operator as a FlockMTL backend. The related-work positioning (Section 9) argues that IOLM-DB is composable beneath semantic-operator systems rather than competing with them: a column-calibrated compressed operator is a drop-in model backend. We demonstrate that composition using vLLM's OpenAI-compatible server. We serve the FP16, 8-bit, and 4-bit checkpoints and run the same FlockMTL semantic query over the same rows against each, pointing FlockMTL's `MODEL` binding at the local endpoint. The interface, query, and engine remain identical, and only the served checkpoint changes. Table 12 reports each backend's footprint and fidelity relative to FlockMTL's FP16 run.

The backend swap reproduces direct-invocation behavior to within 0.015 in every cell. Inside FlockMTL, Fidelity scores 0.997 on Filter and 0.897 on Projection, compared with 0.997 and 0.906 under direct invocation. Compact scores 0.979 and 0.638, compared with 0.987 and 0.653 directly. The host engine is therefore behaviorally transparent to compression, and the same precision guidance applies inside the DuckDB query: Fidelity supports both operator types, while Compact is strongest for categorical Filter outputs. This experiment establishes composability as a measured system property.

Economic break-even. Running a remote API incurs a per-token charge that grows linearly with column cardinality. Local IOLM-DB pays a fixed one-time generation cost C_{gen} in dollars (generation GPU-hours multiplied by the accelerator rate) plus a per-row inference cost proportional to the cloud GPU rate r. The break-even cardinality is:

$$N^* = \frac{C_{\text{gen}}}{c_{\text{API}} - r/(3600\lambda)} \tag{6}$$

where c_{API} is the API cost per row and λ is the operator's measured throughput in rows per second. The factor 3600 converts the GPU rate r from dollars per hour to dollars per second.

We use the fuzzy-join workload (46 input tokens, 2 output tokens per row) for this analysis because it minimizes per-row API charges, putting the API in its strongest economic position. Applied to summarization or correction, whose larger token budgets raise the per-row API cost, the same calculation would yield lower break-even cardinalities and larger savings, so the numbers below are a conservative bound on the gap between local and API execution. Table 13 shows that IOLM-DB reaches API cost parity at column cardinalities between 410 and 6600 rows, far below the tens-of-thousands- to million-row column-scale workloads that instance optimization targets. The LMS student breaks even at 410 rows because its training cost is small, and IOLM-DB-Fidelity breaks even at 6600 rows. IOLM-DB-Compact, whose 128-row-calibrated GPTQ run takes roughly 16 min, breaks even at 6100 rows. Beyond N^* every additional row is processed at approximately $r/(3600\lambda) \approx \$0.002/1\text{k}$ rows, almost two orders of magnitude below the API rate. At one million rows, LMS costs \$0.88 against \$135.00 for GPT-4o, and Compact costs \$2.61. These savings are structural: the per-token API charge is a variable cost that grows without bound, while the local marginal cost is bounded by GPU compute and shrinks with throughput.

Operator co-residency. A recurring analytical workload often applies several semantic operators in sequence over different columns. Without co-residency each operator swap requires evicting the prior model from GPU memory and loading the next, adding model-load latency to every inter-operator transition. Table 14 reports measured GPU memory for the three distinct column-calibrated IOLM-DB-Compact checkpoints (one per workload) loaded simultaneously on the A100.

The co-residency result translates footprint reduction into an execution-plan benefit. Three Compact checkpoints occupy 15.93 GB, compared with 44.87 GB for three FP16 copies, leaving approximately 29 GB of additional A100 headroom for key–value caches, batching buffers, or concurrent work. Under smaller budgets, the same difference becomes a feasibility boundary.

Table 19 measures a synthetic steady-state three-forward-pass probe after the operators have been loaded once; it is not an end-to-end data-bearing analytical query. Sequential FP16 execution incurs 20.0 s of model-swap overhead per probe pass, whereas the co-resident Compact plan removes that overhead and completes in 4.9 s rather than 20.5 s. The 4.9 s Compact inference time, compared with 0.5 s for FP16, reflects quantization-kernel overhead on these very short probes. The fixed swap saving must be weighed against Compact's lower A100 row throughput, so this probe does not establish a universal latency win or a row-count crossover for a heterogeneous multi-operator query. Table 2 gives the full-workload rates; an end-to-end crossover requires the row count and throughput of each operator in the actual plan.

Table 13
Economic break-even between GPT-4o API calls and local IOLM-DB execution on the fuzzy join workload (46 input tokens, 2 output tokens per row; Eq. (6)). C_{gen} is the one-time compression or training cost obtained by pricing the measured 17.5, 16.3, and 1.1 min generation times in Table 1 at a spot A100 rate of $3.00/GPU-hour. The pricing snapshot uses GPT-4o rates of $2.50/1M input tokens and $10.00/1M output tokens (OpenAI model documentation, accessed August 2026: https://developers.openai.com/api/docs/models/gpt-4o). N^* is the column cardinality at which local execution becomes cheaper than API calls; all three local variants are cheaper than the API for any column larger than N^* rows.

Variant	C_{gen}	λ (rows/s)	Marginal cost (per 1k rows)	Break-even N^*	Savings at 1M rows
GPT-4o (API)	–	–	$0.135	–	–
IOLM-DB-Fidelity (8-bit)	$0.88	522.1	$0.002	6 600 rows	98.2%
IOLM-DB-Compact (4-bit)	$0.82	463.9	$0.002	6 100 rows	98.1%
LMS (T5-Small)	$0.06	1007.1	$0.001	410 rows	99.3%

Table 14
Measured resident GPU memory (GB) for N concurrently resident operator checkpoints on the A100 80 GB, read from the device allocator; the 14.96 GB single-FP16 figure is this measured allocation and corresponds to the 14.98 GB weight footprint cited elsewhere (the small difference is allocator rounding). Each compressed checkpoint is a distinct column-calibrated artifact (summarization, data correction, fuzzy join). Three 4-bit operators co-reside in 15.93 GB, approximately the footprint of a single FP16 checkpoint (14.96 GB), whereas three FP16 copies require 44.87 GB.

# Operators	Variant	Cumulative GPU mem.	Marginal per operator
1	FP16 (Llama-3.1-8B)	14.96 GB	14.96 GB
2	FP16	29.92 GB	14.96 GB
3	FP16	44.87 GB	14.96 GB
1	4-bit (IOLM-DB-Compact)	5.31 GB	5.31 GB
2	4-bit	10.62 GB	5.31 GB
3	4-bit	15.93 GB	5.31 GB

Table 15
CBOP choice versus the post-hoc oracle across six measured regimes. The profile-source column identifies the controlled harness within which candidates are compared; throughput values are never compared across rows or harnesses. L40S rows use hardware throughput from Table 10 and checkpoint fidelity from the Products precision profile. Five implementations win at least once: batching, a native 3B model, Fidelity, W8A8, and Compact. Fidelity values are normalized to the reference; "only fit" means the only candidate satisfying both memory and fidelity constraints. CBOP matches the oracle in all six regimes.

Regime	GPU/operator	Memory budget	α_{min}	N	Profile source	CBOP picks	Oracle	Match
A	A100/Projection	80 GB	0.90	1M	PySpark/Movies	Batch FP16	Batch (91.4 r/s, 0.99)	✓
B	A100/Filter	80 GB	0.85	1M	Native-SLM/Movies+Products	Native 3B	Native 3B (652 r/s, 0.89)	✓
C	L40S/Projection	14.8 GB	0.90	1M	L40S/Products	Fidelity	Fidelity (only quality-fit)	✓
D	A100/Projection	80 GB	0.75	1M	Precision sweep/Products	W8A8	W8A8 (206.1 r/s, 0.785)	✓
E	A100/Projection	5.5 GB	0.60	1M	Precision sweep/Products	Compact	Compact (only fit)	✓
F	L40S/Projection	22.2 GB	0.65	1M	L40S/Products	Compact	Compact (160.1 r/s, 0.721)	✓

5.6. CBOP as a decision procedure: Choices vs. the oracle

The preceding subsections measure each strategy in isolation. Here we evaluate CBOP end to end. For each regime, the planner receives the operator type, row count, fidelity floor, available memory, and hardware-specific profile. It applies the feasibility and cost model from Section 3.4 and returns one implementation. We compare that choice with an oracle computed post hoc from the complete candidate measurements in Section 5.3 and Tables 7, 10, and 11.

The 6/6 agreement is a consistency check rather than an out-of-sample accuracy claim: the post-hoc oracle and CBOP consume the same measured profiles. To expose the actual decision boundaries, Table 16 sweeps the two constraints that users set. The selected implementation changes at measured footprint and fidelity boundaries rather than at one hard-coded memory threshold.

We additionally hold out the complete L40S 14.8 GB throughput profile. CBOP uses only the independently measured 22.2 GB L40S profile, applies a 14.8 GB memory budget, $\alpha_{min} = 0.65$, and $N = 10^6$, and predicts Compact. Scoring that choice against the held-out 14.8 GB measurements also selects Compact; its predicted throughput is 160.1 rather than the observed 159.0 rows/s, a 0.7% error. This check is limited to a second memory cap on the same GPU and does not establish transfer across architectures, but it avoids using the target regime to make its own prediction.

Table 16
CBOP sensitivity on A100 Products/Projection at $N = 10^6$. Each transition is induced by a measured candidate's footprint or fidelity. "Infeasible" means that no profiled candidate satisfies both constraints, so the planner returns no plan rather than silently violating the quality floor.

Memory budget	Fidelity floor α_{min}	CBOP choice
5.5 GB	≤ 0.721/> 0.721	Compact/infeasible
6.0 GB	≤ 0.747/> 0.747	Mixed/infeasible
9.0 GB	≤ 0.785/$(0.785, 0.953]$/> 0.953	W8A8/Fidelity/infeasible
15.0 GB	≤ 0.785/> 0.785	W8A8/Direct FP16

Profiling error can still produce a wrong plan. In regime F, Compact beats Direct FP16 only while its profiled throughput exceeds 142.2 rows/s after its 978-second generation cost is charged. That boundary is 11.2% below the measured 160.1 rows/s; an underestimate beyond it makes CBOP conservatively choose Direct FP16, while an overestimate can choose Compact when its realized end-to-end time is worse. The system therefore records realized throughput and refreshes a stale profile rather than treating the cost-model output as a correctness guarantee.

Together, the decision matrix and sensitivity analysis establish that CBOP is more than a memory threshold. At 80 GB, it selects batching for high-fidelity Projection, a native 3B model for Filter, and W8A8 for quality-relaxed Projection. Under a 14.8 GB L40S cap, the 0.90

floor selects Fidelity. A 5.5 GB artifact budget makes Compact the only feasible implementation, while at 22.2 GB on the L40S Compact wins on measured execution cost once N exceeds its finite break-even. The complete strategy profiles, sweep, held-out check, and error boundary are reported in the revised evaluation.

6. Implementation and deployment considerations

To invoke optimized operators inside tabular execution systems, the prototype exposes a narrow runtime contract. We separate that contract from the empirical study because it should be portable across engines.

Prototype boundary. The Python prototype uses pandas DataFrames for tabular execution, vLLM for local inference, Hugging Face Transformers for model loading and fine-tuning, and CUDA for GPU execution. The optimized operator is a model artifact plus an invocation policy and does not depend on pandas.

Two implementation optimizations can further improve the prototype. Reducing vLLM cache reservation would permit more concurrent operator instances, while a compiled DataFrame boundary would remove Python transfer overhead. Neither changes the model artifact or planner design evaluated here.

6.1. UDF interface and execution semantics

IOLM-DB exposes semantic operators as user-defined functions (UDFs) compatible with SQL query processors. A representative scalar signature is:

```
llm_prompt(
  prompt_template TEXT,
  input_text TEXT,
  optimization_hint TEXT DEFAULT 'balanced',
  min_quality FLOAT DEFAULT 0.95,
  timeout_ms INT DEFAULT 10000
) RETURNS TEXT
```

Join-like predicates use the same contract with two input text columns and a boolean output.

The prototype implements the operator as a scalar UDF together with the two-column join-predicate variant above, and aggregate semantic operators are outside its current scope. It is a single-process Python component, not a compiled in-engine function, that delegates intra-GPU batching to vLLM. Inputs longer than the configured model context (4096 tokens in our experiments) are truncated and flagged as length-OOD inputs for the fallback path of Section 7. Compiling this contract into an in-process engine binding is engineering work that leaves the method unchanged.

Execution semantics are explicit:

- **NULL handling:** if input columns are NULL, output is NULL and no model call is issued.
- **Determinism mode:** reproducibility-sensitive runs can use a fixed decoding seed.
- **Timeout policy:** on timeout, the operator either fails fast or falls back to baseline execution based on the query option.
- **Quality guardrail:** when enabled, anomalous outputs trigger row-level fallback to the reference endpoint.
- **Cache key:** tuple of prompt template, normalized input, model version, and optimization profile.

The UDF contract is portable across analytical engines. Bindings for PostgreSQL [21], Spark [22], and ClickHouse [23] are straightforward implementation extensions to the evaluated prototype.

Monitoring and observability. The runtime records fallback rate, cache hit rate, per-row latency, throughput, GPU memory use, selected optimization profile, calibration-set identifier, and generation-time components. These logs support two operational checks: whether the optimized operator still matches the expected input distribution, and whether the optimization overhead is amortizing as predicted by CBOP.

6.2. Reproducibility checklist

For reproducibility, each experiment configuration has a checklist entry containing:

- dataset source and preprocessing script version,
- prompt template and decoding parameters,
- calibration/validation sampling settings,
- candidate optimization profile and selected model variant,
- software stack versions (CUDA, vLLM, transformers),
- hardware profile and random seed configuration.

The checklist lets peers reconstruct each reported result from a complete configuration record rather than high-level narrative alone.

6.2.0.1. Artifact availability. The source code, deployment scripts, evaluation recipes, datasets, and supporting experimental artifacts are publicly available at https://github.com/mpi-dsg/IOLMDB-JL. The repository includes instructions for reproducing all experiments reported in this paper, together with the compression scripts, calibration sets, bitwidth maps, measured generation-time logs, and the documented phase-apportionment procedure used in Table 1.

7. Limitations and threats to validity

Instance-level optimization exploits the *stationarity* of the target column. Its failure modes follow from that assumption.

Assumption of stable prompts and input distributions. The pipeline assumes that the prompt template is fixed across the full column and that the distribution of row contents is approximately stationary over the window covered by the calibration sample. This assumption fits many tabular workloads, such as summarizing all reviews in a product table, correcting typos across a code column, or matching entities across two supplier lists. It is a poor fit for ad-hoc exploratory workflows that vary the prompt per row and for columns that drift over time, such as a reviews column whose product mix changes seasonally. In those cases, users should not expect a single instance-optimized operator to match the reference model.

Out-of-distribution (OOD) inputs. The main failure mode is a row whose content is not represented by the calibration sample. During development we observed three OOD regimes:

- *Lexical OOD.* Inputs whose vocabulary or entity types are absent from calibration (e.g., an English electronics-review column later populated with French reviews or a new product category). These shifts reduce fidelity when the calibration set no longer represents the deployed column.
- *Structural OOD.* Inputs that break the structural assumptions of the prompt template (e.g., a row that is a JSON blob rather than free-form text, or an entity-match input that contains nulls on one side of the concatenation). Here the compressed operator behaves less predictably than the reference model and failures manifest as empty outputs, repeated tokens, or pattern-violating responses.
- *Length OOD.* Inputs much longer than anything seen in calibration. Quantization error accumulates with sequence length, and attention projections quantized to 8-bit precision on a short-input calibration set can still produce activation outliers on long inputs.

The fallback mechanism addresses detectable cases in all three regimes:

when an output fails a configurable check (empty response, pattern mismatch, or length anomaly), the runtime routes that row to a local or remote reference endpoint. A sustained increase in the fallback rate triggers a new sample C and operator regeneration. Developing low-overhead detectors for subtler distribution shifts remains an important extension.

Extreme outliers. Even within a broadly stationary distribution, individual rows with extreme lengths or atypical content can trigger failure. Quantization error in transformer models is often driven by activation outliers, and the sensitivity-based bitwidth assignment in Section 3.2.1 can only see outlier behavior present in the calibration set. The reserved outlier slice in Section 3.2.1 is therefore a defensive measure, not a guarantee. Production deployments should log and periodically re-audit rows whose fallback rate is elevated.

Heterogeneous column formats. A separate failure mode arises when a single column is *structurally* heterogeneous, for example when it mixes free text, semi-structured blobs, abbreviations, and multilingual content. A single instance-optimized operator is a poor fit for such columns because the calibration set is drawn from a mixture distribution, and the resulting compression compromises across modes instead of matching any one mode well. For these columns, two architectural remedies are possible but not yet implemented. The system could *(a)* pre-cluster the column and train a small ensemble of per-cluster operators dispatched by a lightweight classifier at runtime, or *(b)* fall back to the uncompressed reference model for underrepresented modes. A single compressed operator is appropriate only when one dominant column mode approximates most rows.

When the approach helps, and when It does not. The approach helps when one fixed prompt runs over a roughly stationary column and either (i) memory fit or co-residency makes footprint binding or (ii) sufficient row volume amortizes a hardware-specific throughput gain. Direct invocation suits small, one-off columns, batched FP16 is preferred for large short-output workloads on an unconstrained A100, and the operator must be recalibrated when the prompt or the column distribution changes. On lower-bandwidth hardware, the same short-output Compact operator can provide both capacity and throughput gains (Section 5.5). Generative operators should use Fidelity or the mixed profile when Compact falls below the required fidelity floor.

Optimization amortization. Amortization is hardware-dependent. On the A100, the Compact profile does not exceed FP16 throughput, so its throughput break-even is unbounded and CBOP selects it only for memory fit or co-residency. On the L40S, Compact reaches 1.28× throughput and repays its 16.3-minute generation cost at approximately 554,000 rows. Small, one-off transformations that impose no memory constraint remain direct-invocation workloads.

Hardware dependency. The current compressed-tensors W4A16/W8A16 kernels require GPU compute capability ≥ 7.5. Section 5.5 verifies both sides of this boundary: the checkpoints run on the cc 8.9 L40S and are rejected on the cc 7.0 V100. T4, L4, A10, and RTX 20/30/40-series devices satisfy the published capability requirement, but compatibility and absolute throughput still depend on their individual kernel implementations and hardware balance. We did not measure a consumer card directly. The hardware analysis states what the L40S result establishes—a measured capacity and throughput result on one Ada card—and what it does not, namely serviceability or a specific throughput ratio on every card in that class. The present implementation therefore does not serve compressed artifacts on V100-class or CPU-only hosts.

Reference-fidelity measurement. Reference fidelity measures replacement behavior, whereas labeled accuracy measures task quality. Section 5.2 reports both on the labeled Filter workloads and shows that the compressed profiles remain within 0.006 of the reference model's absolute accuracy. Generative Projection has no fixed gold output, so we use ROUGE-L against the reference output. Direct human evaluation would answer a complementary question about acceptable paraphrases and perceived quality.

Scope of operator. Our pipeline optimizes a *single* semantic operator per pass. Multi-operator pipelines that chain several LLM calls (e.g., extract entities, then classify, then summarize) would require separate optimization for each stage. We have not yet studied interactions between compressed operators in a pipeline, including error propagation and calibration of intermediate distributions. The co-residency analysis of Section 5.5 establishes that multiple specialized operators fit in the memory headroom freed by compression, but the end-to-end measurement of a multi-operator analytical query plan is left to future work.

Calibration-set size. The main results use 128 calibration rows. On Products/Projection, increasing the set to 1024 changes fidelity by at most 0.022 while increasing generation time from 19 to 63 min (Table 18). This experiment supports the 128-row operating point for the evaluated distribution. A substantially different domain should repeat the sweep.

Native small-model comparison. Section 5.4 compares the compressed 8B operator with uncompressed native 3B models. The resulting operating points are complementary: native 3B models lead Filter throughput, while the compressed 8B provides higher Projection fidelity at a smaller footprint. Applying column calibration to a 3B base is a promising additional point on this frontier.

Long-output operators. The evaluated Projection operator emits at most 50 words. Longer outputs increase autoregressive decoding and key–value-cache pressure, potentially shifting the A100 into a bandwidth-bound regime. The L40S result supports the underlying mechanism: lowering the bandwidth-to-compute ratio turns A100 parity into a 1.28× gain. A generation-length sweep is identified as a useful follow-up experiment.

API pricing. The economic analysis uses the August 2026 GPT-4o price snapshot ($2.50 per 1M input tokens and $10.00 per 1M output tokens) and a $3.00 A100 GPU-hour. Prices are parameters in Eq. (6), so readers can substitute a different API or accelerator rate without changing the analysis.

8. Supporting measurements

This section collects three supporting measurements referenced in Section 5: the batch-size sweep behind the batching strategy, the calibration-set-size sweep that fixes $|C| = 128$, and the steady-state pipeline latency behind the co-residency result.

9. Related work

IOLM-DB relates to semantic operators, expensive-function optimization, model compression, distillation, and model serving. It specializes the model behind one repeated operator for a fixed prompt and target column.

Semantic operators and database integration. Foundation models have been used for data wrangling and semantic tabular transformations [5]. LOTUS exposes semantic operators over DataFrames with optimizer support and accuracy guarantees [6]. FlockMTL embeds model-driven scalar and aggregate functions in DuckDB and supports batching and caching over unmodified model backends [7,8]. Palimpzest optimizes declarative pipelines of model-powered operators across cost-quality trade-offs [9]. DocETL rewrites document-processing pipelines with agent-guided plan evaluation to improve output accuracy [24], and CAESURA uses an LLM to plan multi-modal query execution over

Table 17
Batch-processing results by workload and batch size. The table reports total runtime, row throughput, and average input/output tokens per row.

Workload	Batch Size	Total Time (s)	Throughput (rows/s)	Tokens/Row (In/Out)
Amazon Reviews	10	228.12	6.58	78.3/11.6
	20	216.70	6.92	75.1/11.0
	50	226.87	6.61	73.1/11.3
GitHub Typos	10	228.07	6.58	30.0/12.0
	20	224.66	6.68	26.8/11.8
	50	217.31	6.90	24.9/11.4
Fuzzy Join	10	88.56	16.94	36.7/4.4
	20	79.70	18.82	34.3/4.0
	50	84.68	17.71	32.8/4.3

Table 18
Effect of calibration-set size on 4-bit Products/Projection under a fixed sampling seed. Fidelity remains within 0.687–0.709 across an 8× range of $|C|$, while generation time increases from 19 to 63 min.

Calibration size $\|C\|$	Fidelity (4-bit Projection)	Generation time (min)
128	0.695	18.9
256	0.705	23.7
512	0.709	38.2
1024	0.687	62.8

Table 19
Synthetic three-operator probe latency in *steady state*: serial model swapping (FP16) versus co-resident execution (4-bit), measured after all operators have been loaded once. Probe inference is one forward pass per operator at `max_new_tokens` = 4; this is not an end-to-end data-bearing query. Section 5.5 discusses the trade-off between fixed swap savings and workload-specific per-row throughput.

Variant	Swap overhead (3 ops)	Probe inference (3 ops)	Pipeline total
FP16 (sequential swap)	20.0 s	0.5 s	20.5 s
4-bit (co-resident)	0 s	4.9 s	4.9 s

tables, text, and images [25]. TAG proposes table-augmented generation as a unified paradigm that combines database retrieval with natural-language reasoning for analytics beyond what Text2SQL expresses [26], and UQE extends query execution to unstructured collections by combining LLM calls with compiler-inspired sampling and optimization [27]. ELEET targets per-call cost directly by replacing general-purpose LLMs with a pre-trained small model for query execution over text and tables [28], whereas IOLM-DB specializes an operator's existing reference model to a fixed prompt and target column rather than training a new architecture. Recent work on relational LLM analytics reduces query cost by reordering rows and fields to maximize key–value-cache reuse across calls [29]. NeurDB studies database and neural-model co-design [30], and broader surveys describe database management system integration with LLMs. IOLM-DB is complementary: it does not introduce a new query interface or pipeline planner; it reduces the resident footprint and, in bandwidth-bound regimes, the execution cost of the model artifact behind an operator. Table 20 summarizes the closest composable systems.

The three systems optimize different, composable layers. LOTUS controls how semantic operators are expressed and planned over DataFrames. FlockMTL embeds model calls as DuckDB functions and adds batching and caching. IOLM-DB changes the model artifact behind one recurring operator. Consequently, a column-calibrated checkpoint can serve as the local backend for either host without changing the query interface. Section 5.5 demonstrates this composition inside FlockMTL and shows that backend behavior remains within 0.015 of direct invocation. Host-level planning reduces the number and scheduling cost of calls, while IOLM-DB reduces the memory footprint and, on bandwidth-limited hardware, the execution cost of each remaining call.

Expensive functions and data wrangling. Database optimizers have long treated expensive UDFs as first-class cost drivers. Predicate reordering and short-circuiting can reduce the number of calls when selectivity estimates are available [12]. Program-synthesis systems such as FlashFill++, semantic regular-expression synthesis, and Transform-Data-by-Example infer transformations from examples [31–33]. Code-generation approaches use LLMs to emit database or data-wrangling code [34,35]. These systems reduce calls or synthesize code for narrower transformations. IOLM-DB keeps the LLM-operator abstraction and optimizes the model used by a repeated operator.

Adaptive and learned query optimization. CBOP is an adaptive operator-implementation chooser, inheriting from a long line of database work on selecting execution strategies from observed statistics. Eddies reroute tuples through operators in response to runtime conditions [36], and adaptive query processing formalizes when systems should reoptimize as they learn about their data [37]. More recent systems learn the cost model itself: OtterTune tunes database configurations from observed workloads [38], Neo replaces a hand-built optimizer with execution feedback [39], and BRAD routes queries across cloud engines using predicted performance and operating cost [40]. IOLM-DB also follows the broader instance-optimized-systems principle of adapting a component to a particular workload and data distribution [41]; here the specialized component is the model artifact behind one semantic operator. CBOP applies this principle to the decision variable *which compiled implementation of a semantic operator to serve*: direct, batched, native-small, weight-quantized, activation-quantized, mixed-precision, or distilled, using measured generation cost, throughput, fidelity, and memory fit. Section 5.6 evaluates these choices directly against a post-hoc oracle.

Compression and distillation. Quantization, pruning, and distillation reduce the cost of large language models. GPTQ and SmoothQuant lower precision while controlling quantization error and activation outliers [13,14]. AWQ scales salient weight channels identified from activation statistics, enabling low-bit weight-only quantization without backpropagation [19], and QuaRot applies randomized Hadamard rotations to remove activation outliers and enable end-to-end 4-bit inference [42]. SparseGPT and LLM-Pruner remove weights or transformer structures with low measured importance [15,16]. DistilBERT and T5 show that smaller student models can preserve useful behavior from larger teachers under task-specific training [17,18]. IOLM-DB contributes the system-level specialization layer above these compression algorithms. It exploits a database-specific opportunity — the prompt and target-column distribution are known before full execution — to construct workload-aware calibration data, profile multiple operator implementations, and expose model footprint, fidelity, and generation cost to the query planner.

Table 20
Closest systems for LLM semantic operators over tabular data. IOLM-DB optimizes the model behind one column-scale operator; FlockMTL integrates model functions into DuckDB; LOTUS exposes semantic operators over DataFrames. Palimpzest [9] operates at the pipeline-planning layer and is therefore discussed in the text rather than placed in this feature table.

Feature	IOLM-DB	FlockMTL	LOTUS
Model compression	✓	–	–
Knowledge distillation	✓	–	–
Column-specific optimization	✓	–	–
Calibration-set methodology	✓	–	–
Batching and caching	✓	✓	–
Semantic operators	Single task	Scalar and aggregate	DataFrame operators
Host runtime	SQL UDFs, DataFrame	DuckDB	pandas DataFrames
Model backends	Local models	Ollama, OpenAI	OpenAI, local models
Open source artifact	✓	✓	✓

Serving and batching. Serving systems improve LLM throughput through batching, scheduling, memory management, and key–value-cache reuse. Orca introduces iteration-level scheduling and selective batching for transformer inference [43]. vLLM uses PagedAttention to improve serving memory efficiency for large language models [44], and SGLang adds RadixAttention to reuse the key–value cache across structured calls in a single program [45]. These systems accelerate unmodified models across mixed request streams. IOLM-DB also uses batching, while its main optimization creates a smaller operator for a recurring workload and uses the planner to decide when the one-time optimization cost amortizes.

10. Conclusion

IOLM-DB compiles a recurring prompt–column pair into a resource-right-sized semantic operator. Column calibration improves 4-bit GPTQ fidelity in all six evaluated workload–operator cells, and output-aware calibration provides a further gain in every generative Projection cell. Quantization reduces the resident Llama-3.1-8B footprint by 1.8× at 8-bit and 2.8× at 4-bit. The Fidelity profile preserves 0.93–1.00 of reference behavior, and Compact preserves 0.96–0.98 on Movies and Products Filter. The deployment effect is equally concrete. Three Compact operators co-reside in 15.93 GB, both compressed profiles run under a 14.8 GB cap where FP16 cannot initialize, and Compact reaches 1.28× FP16 throughput on the lower-bandwidth L40S. CBOP integrates these measurements with generation cost, fidelity, and memory constraints and matches the post-hoc oracle in all six evaluated regimes. These results establish model footprint as an optimizer-visible property of a semantic operator and show that a fixed LLM transformation can be deployed as a compact, hardware-aware artifact rather than an immutable call to a general-purpose backend.

CRediT authorship contribution statement

Bardia Mohammadi: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Data curation, Conceptualization. **Laurent Bindschaedler:** Writing – review & editing, Writing – original draft, Supervision, Software, Project administration, Methodology, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: Advances in Neural Information Processing Systems, vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.

[2] Gemini Team, Google, Gemini: A family of highly capable multimodal models, 2023, http://dx.doi.org/10.48550/arXiv.2312.11805, arXiv preprint arXiv:2312.11805.

[3] Llama Team, AI @ Meta, The Llama 3 herd of models, 2024, http://dx.doi.org/10.48550/arXiv.2407.21783, arXiv preprint arXiv:2407.21783.

[4] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y.T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M.T. Ribeiro, Y. Zhang, Sparks of artificial general intelligence: Early experiments with GPT-4, 2023, http://dx.doi.org/10.48550/arXiv.2303.12712, arXiv preprint arXiv:2303.12712.

[5] A. Narayan, I. Chami, L. Orr, C. Ré, Can foundation models wrangle your data? Proc. the VLDB Endow. 16 (4) (2022) 738–746, http://dx.doi.org/10.14778/3574245.3574258.

[6] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, M. Zaharia, Semantic operators and their optimization: Enabling LLM-based data processing with accuracy guarantees in LOTUS, Proc. the VLDB Endow. 18 (11) (2025) 4171–4184, http://dx.doi.org/10.14778/3749646.3749685.

[7] A. Dorbani, S. Yasser, J. Lin, A. Mhedhbi, Beyond quacking: Deep integration of language models and RAG into DuckDB, Proc. the VLDB Endow. 18 (12) (2025) 5415–5418, http://dx.doi.org/10.14778/3750601.3750685.

[8] M. Raasveldt, H. Mühleisen, DuckDB: an embeddable analytical database, in: Proceedings of the 2019 International Conference on Management of Data, Association for Computing Machinery, 2019, pp. 1981–1984, http://dx.doi.org/10.1145/3299869.3320212.

[9] C. Liu, M. Russo, M. Cafarella, L. Cao, P.B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, G. Vitagliano, Palimpzest: Optimizing AI-powered analytics with declarative query processing, in: Proceedings of the 15th Annual Conference on Innovative Data Systems Research, CIDR'25, 2025, URL https://www.vldb.org/cidrdb/papers/2025/p12-liu.pdf.

[10] J. Ni, J. Li, J. McAuley, Justifying recommendations using distantly-labeled reviews and fine-grained aspects, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP, Association for Computational Linguistics, Hong Kong, China, 2019, pp. 188–197, http://dx.doi.org/10.18653/v1/D19-1018.

[11] M. Hagiwara, M. Mita, GitHub typo corpus: A large-scale multilingual dataset of misspellings and grammatical errors, in: Proceedings of the Twelfth Language Resources and Evaluation Conference, European Language Resources Association, Marseille, France, 2020, pp. 6761–6768.

[12] J.M. Hellerstein, Optimization techniques for queries with expensive methods, ACM Trans. Database Syst. 23 (2) (1998) 113–157, http://dx.doi.org/10.1145/292481.277627.

[13] E. Frantar, S. Ashkboos, T. Hoefler, D. Alistarh, OPTQ: Accurate quantization for generative pre-trained transformers, in: The Eleventh International Conference on Learning Representations, 2023, URL https://openreview.net/forum?id=tcbBPnfwxS.

[14] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, S. Han, SmoothQuant: Accurate and efficient post-training quantization for large language models, in: Proceedings of the 40th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 38087–38099.

[15] E. Frantar, D. Alistarh, SparseGPT: Massive language models can be accurately pruned in one-shot, in: Proceedings of the 40th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 10323–10337.

[16] X. Ma, G. Fang, X. Wang, LLM-Pruner: On the structural pruning of large language models, in: Advances in Neural Information Processing Systems, vol. 36, Curran Associates, Inc., 2023, pp. 21702–21720.
[17] V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, in: 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing (EMC^2) At NeurIPS 2019, 2019, arXiv:1910.01108.
[18] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P.J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, J. Mach. Learn. Res. 21 (140) (2020) 1–67.
[19] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, S. Han, AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration, vol. 6, 2024, URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.
[20] A. Alaparthi, P. Loh, R. Marcus, ScaleLLM: A technique for scalable LLM-augmented data systems, in: Companion of the 2025 International Conference on Management of Data, Association for Computing Machinery, 2025, pp. 11–14, http://dx.doi.org/10.1145/3722212.3725130.
[21] B. Momjian, PostgreSQL: Introduction and Concepts, Addison-Wesley, Boston, MA, ISBN: 9780201703313, 2001.
[22] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, M. Zaharia, Spark SQL: Relational data processing in Spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, 2015, pp. 1383–1394, http://dx.doi.org/10.1145/2723372.2742797.
[23] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, A. Milovidov, ClickHouse - Lightning fast analytics for everyone, Proc. the VLDB Endow. 17 (12) (2024) 3731–3744, http://dx.doi.org/10.14778/3685800.3685802.
[24] S. Shankar, T. Chambers, T. Shah, A.G. Parameswaran, E. Wu, DocETL: Agentic query rewriting and evaluation for complex document processing, Proc. the VLDB Endow. 18 (2025) http://dx.doi.org/10.14778/3746405.3746426.
[25] M. Urban, C. Binnig, CAESURA: Language models as multi-modal query planners, in: 14th Conference on Innovative Data Systems Research, CIDR 2024, 2024, URL https://www.cidrdb.org/cidr2024/papers/p14-urban.pdf.
[26] A. Biswal, L. Patel, S. Jha, A. Kamsetty, S. Liu, J.E. Gonzalez, C. Guestrin, M. Zaharia, Text2SQL is not enough: Unifying AI and databases with TAG, in: 15th Conference on Innovative Data Systems Research, CIDR 2025, 2025, URL https://vldb.org/cidrdb/papers/2025/p11-biswal.pdf.
[27] H. Dai, B.Y. Wang, X. Wan, B. Dai, S. Yang, A. Nova, P. Yin, P.M. Phothilimthana, C. Sutton, D. Schuurmans, UQE: A query engine for unstructured databases, in: Advances in Neural Information Processing Systems, vol. 37, Curran Associates, Inc., 2024.
[28] M. Urban, C. Binnig, ELEET: Efficient learned query execution over text and tables, Proc. the VLDB Endow. 17 (13) (2024) 4867–4880, http://dx.doi.org/10.14778/3704965.3704989.
[29] S. Liu, A. Biswal, A. Kamsetty, A. Cheng, L.G. Schroeder, L. Patel, S. Cao, X. Mo, I. Stoica, J.E. Gonzalez, M. Zaharia, Optimizing LLM queries in relational data analytics workloads, in: Proceedings of Machine Learning and Systems, vol. 7, 2025, URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/b5dc49f44db2fadc5c4d717c57f4a424-Abstract-Conference.html.
[30] Z. Zhao, S. Cai, H. Gao, H. Pan, S. Xiang, N. Xing, G. Chen, B.C. Ooi, Y. Shen, Y. Wu, M. Zhang, NeurDB: On the design and implementation of an AI-powered autonomous database, in: Proceedings of the 15th Annual Conference on Innovative Data Systems Research, CIDR'25, 2025, URL https://www.vldb.org/cidrdb/papers/2025/p29-zhao.pdf.
[31] J. Cambronero, S. Gulwani, V. Le, D. Perelman, A. Radhakrishna, C. Simon, A. Tiwari, FlashFill++: Scaling programming by example by cutting to the chase, Proc. the ACM Program. Lang. 7 (POPL) (2023) 952–981, http://dx.doi.org/10.1145/3571226.
[32] Q. Chen, A. Banerjee, Ç. Demiralp, G. Durrett, I. Dillig, Data extraction via semantic regular expression synthesis, Proc. the ACM Program. Lang. 7 (OOPSLA2) (2023) 1848–1877, http://dx.doi.org/10.1145/3622863.
[33] Y. He, X. Chu, K. Ganjam, Y. Zheng, V.R. Narasayya, S. Chaudhuri, Transform-data-by-example (TDE): An extensible search engine for data transformations, Proc. the VLDB Endow. 11 (10) (2018) 1165–1177, http://dx.doi.org/10.14778/3231751.3231766.
[34] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. Petroski Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. Hebgen Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A.N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code, 2021, http://dx.doi.org/10.48550/arXiv.2107.03374, arXiv preprint arXiv:2107.03374.
[35] X. Li, T. Döhmen, Towards efficient data wrangling with LLMs using code generation, in: Proceedings of the Eighth Workshop on Data Management for End-To-End Machine Learning, Association for Computing Machinery, 2024, pp. 62–66, http://dx.doi.org/10.1145/3650203.3663334.
[36] R. Avnur, J.M. Hellerstein, Eddies: Continuously adaptive query processing, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000, pp. 261–272, http://dx.doi.org/10.1145/342009.335420.
[37] A. Deshpande, Z. Ives, V. Raman, Adaptive query processing, Found. Trends Databases 1 (1) (2007) 1–140, http://dx.doi.org/10.1561/1900000001.
[38] D. Van Aken, A. Pavlo, G.J. Gordon, B. Zhang, Automatic database management system tuning through large-scale machine learning, in: Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, 2017, pp. 1009–1024, http://dx.doi.org/10.1145/3035918.3064029.
[39] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, Proc. the VLDB Endow. 12 (11) (2019) 1705–1718, http://dx.doi.org/10.14778/3342263.3342644.
[40] T. Kraska, T. Li, S. Madden, M. Markakis, A. Ngom, Z. Wu, G.X. Yu, Check out the big brain on BRAD: Simplifying cloud data processing with learned automated data meshes, Proc. the VLDB Endow. 16 (11) (2023) 3293–3301, http://dx.doi.org/10.14778/3611479.3611526.
[41] T. Kraska, Towards instance-optimized data systems, Proc. the VLDB Endow. 14 (12) (2021) 3222–3232, http://dx.doi.org/10.14778/3476311.3476392.
[42] S. Ashkboos, A. Mohtashami, M. Croci, B. Li, P. Cameron, M. Jaggi, D. Alistarh, T. Hoefler, J. Hensman, QuaRot: Outlier-free 4-bit inference in rotated LLMs, in: Advances in Neural Information Processing Systems, vol. 37, Curran Associates, Inc., 2024.
[43] G.-I. Yu, J.S. Jeong, G.-W. Kim, S. Kim, B.-G. Chun, Orca: A distributed serving system for transformer-based generative models, in: 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 22, USENIX Association, 2022, URL https://www.usenix.org/conference/osdi22/presentation/yu.
[44] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C.H. Yu, J.E. Gonzalez, H. Zhang, I. Stoica, Efficient memory management for large language model serving with PagedAttention, in: Proceedings of the 29th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, 2023, pp. 611–626, http://dx.doi.org/10.1145/3600006.3613165.
[45] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C.H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J.E. Gonzalez, C. Barrett, Y. Sheng, SGLang: Efficient execution of structured language model programs, in: Advances in Neural Information Processing Systems, vol. 37, Curran Associates, Inc., 2024.