Expanding the Physical Design Space of LLM Data Systems

Gabriele Sanmartino[1], Matthias Urban[2], Carsten Binnig[2], and Paolo Papotti[1]
[1]EURECOM, France
[2]TU Darmstadt, Germany

Abstract

Large language models are becoming first-class components of data systems through semantic operators that declaratively process text, images, audio, and other unstructured data. While recent systems have made progress in defining such operators and optimizing their execution, their physical design space remains comparatively narrow: an optimizer often chooses among models, prompts, cascades, or approximate filters, but treats inference itself largely as a black box. This article argues that LLM-native data systems should expose selected inference-time mechanisms to the optimizer as physical design choices. We focus on key–value (KV) caches in transformer decoders as a concrete example. Rather than viewing KV caches only as a serving optimization, we treat them as materializable physical representations of data items that can be precomputed, compressed, reused, and selected during query planning. Different cache profiles induce different trade-offs among latency, memory, cost, and output quality, thereby expanding the set of physical implementations available for semantic operators. To make this larger space manageable, we advocate constructing Pareto frontiers of candidate implementations and exposing only non-dominated choices to the optimizer. We discuss how this design pattern applies to LLM-native systems such as Lotus and Stretto, where selecting the right physical operators can preserve quality guarantees while reducing inference time. The broader message is that LLM-native query engines require an inference-aware physical layer, not merely better prompts or faster model serving.

1 Introduction

Large language models are increasingly becoming first-class components of data systems. Instead of treating text, images, audio, and other unstructured objects as inputs to an external preprocessing pipeline, recent *LLM data systems* expose *semantic operators* that allow users to query such data declaratively. A user may ask for documents that satisfy a natural language (NL) predicate, extract structured attributes from reports, join records according to semantic similarity, or rank multi-modal objects according to a task-specific criterion. Systems such as Palimpzest [9], Lotus [12], DocETL [18], ThalamusDB [5], Galois [17], and Caesura [19] illustrate this shift from LLMs as external tools to LLMs as components of the query execution stack.

This shift raises a familiar question for database systems: what is the physical design space behind a logical query? Classical systems separate the logical meaning of an operator from its physical realization. A join may be implemented as a hash join, sort-merge join, nested-loop join, or index-nested-loop join; the optimizer chooses among them using cost and cardinality estimates. LLM data systems need a similar separation. A semantic filter, map, or join specifies *what* the system should compute, but the system may implement it using different models, prompts, retrieval strategies, cascades, thresholds, batching policies, or approximate methods. The difference is that physical choices in LLM systems affect not only latency and resource consumption, but also output quality. The optimizer must therefore reason about a multi-objective space involving cost, latency, memory, and task-level quality.

The current physical space. Early LLM data systems have defined semantic operators and introduced ideas for reducing the cost of executing them. LOTUS, for example, shows how semantic operators can be optimized using cascades [12]. Other systems explore cost-based search over semantic-operator implementations [15], SQL-style optimizations [17], logical and prompt-level rewrites [14, 21], or quality-preserving routing between cheaper and more expensive LLM calls [22, 23]. These approaches, together with inference-aware techniques such as reduced-context execution, model routing, and serving-level cache management [6, 9, 11], demonstrate that semantic queries should not be executed by naively invoking the largest available model for every tuple.

However, the design space exposed to semantic-query optimizers is still often coarse-grained. A system may choose between a small model and a large LLM, between one prompt and another, between a reduced and a full context, or between invoking an LLM and applying a cheaper embedding-based filter. These choices are useful, and prior work shows that they can reduce inference cost. Our point is complementary: they can still leave gaps along the cost–quality curve when the optimizer lacks intermediate physical implementations. A small LLM or aggressively reduced context may be too inaccurate for a given quality target, while the next available high-quality implementation may be more expensive than necessary. In such settings, the optimizer would benefit from a denser menu of physical operators.

Inference state as physical design. This article argues that LLM data systems should further expand their physical design space by exposing selected inference-time mechanisms to the query optimizer. In particular, we focus on the key–value (KV) cache used by transformer decoders. During autoregressive decoding, the KV cache stores attention keys and values for the already processed context, avoiding repeated recomputation during later decoding steps [13]. In standard serving systems, this cache is primarily managed as a runtime optimization for efficient inference and memory reuse. From a data-management perspective, however, the KV cache can be viewed differently: it is a materializable physical representation of a data item under a given model. Because a cache captures how the model has encoded an input object, it can support repeated semantic processing over that object and can also provide signals for estimating the behavior of semantic operators before they are executed.

This observation changes the role of the cache. If some queries repeatedly condition on the same set of objects—for example, the documents, images, or records—then their KV caches can be precomputed, stored, compressed, and reused across operators and queries. This reuse has two complementary benefits. First, caches can accelerate *processing*: an operator can avoid recomputing the model representation of an input object and can instead condition directly on a cached or compressed representation. Second, caches can improve *planning*: the same representations can be used to estimate which records are likely to satisfy a semantic predicate, how many pairs may survive a semantic join, or which inputs require a more expensive model call. In classical databases, physical design structures such as indexes and materialized views support both efficient access and better optimization. KV caches can play an analogous role for LLM-native systems, acting both as execution artifacts and as statistical objects for semantic cardinality estimation [20].

Moreover, different cache variants can be materialized for the same item. A full cache may preserve high quality but require more memory and lower batching throughput. A compressed cache may reduce memory footprint and accelerate execution, while introducing a controlled degradation in task quality or estimation accuracy. By creating a Pareto frontier of cache profiles, the system obtains multiple physical implementations of the same logical semantic operator, as well as multiple summaries that can support optimization-time estimates. Recent work on KV-cache compression and token pruning provides mechanisms for constructing such variants [3, 4, 8], while LLM serving and routing systems show the importance of inference-time decisions such as cache management, batching, and model selection for efficient inference [6, 10, 11].

KV caches should be exposed to the *optimizer*, not to the user: users should continue to write

declarative queries over semantic operators. The system decides whether an operator instance should use an embedding filter, a full KV cache, or one of the compressed cache profiles. Similarly, it decides when cached representations are useful for estimating selectivities and intermediate result sizes before committing to an execution plan.

From more operators to better choices. Expanding the physical design space is useful only if the optimizer can navigate it. Blindly materializing many cache variants can increase storage cost and make planning harder. A practical system therefore needs a compact representation of the useful choices. We advocate constructing *Pareto frontiers* of physical operator implementations. For each logical semantic operator, or for each class of operator instances, the system profiles candidate implementations along dimensions such as latency and estimated output quality. Dominated implementations are removed: if one implementation is both slower and lower-quality than another, it should not be presented to the optimizer. The remaining frontier gives the optimizer a compact menu of non-dominated physical alternatives.

This frontier-based view makes the system architecture modular. The mechanism that creates physical alternatives—for example, KV-cache compression—can be separated from the mechanism that chooses among them. A rule-based optimizer, a classical cost-based optimizer, or a gradient-based optimizer can all consume a frontier of candidate implementations, provided the frontier exposes the relevant cost and quality estimates.

Physical design for LLM-native data systems. In this paper, we do not argue for an optimizer, a single cache-compression method, or a new semantic-operator API. Instead, we identify a systems pattern that is likely to recur as LLMs become part of data-processing engines: logical semantic operators should be paired with a rich set of physical implementations; inference state, including KV caches, can be materialized and managed as part of that physical space; and Pareto frontiers can make the resulting space usable by query optimizers.

The remainder of the paper develops this argument. We start by giving an example of the role of KV cache in semantic processing. We then describe how physical operator choices arise in LLM data systems and why the cost–quality dimension makes them different from classical physical operators. Next, we introduce KV caches as reusable physical representations of data items and explain how compressed cache profiles expand the operator space. We then discuss how Pareto frontiers can be constructed and used to select among cache-aware and non-cache-aware implementations. Finally, we use systems such as Lotus and Stretto to illustrate the practical impact of this design: by selecting the right physical operators, an engine can preserve quality guarantees while reducing inference time.

2 A Running Example: Cache-Aware Semantic Processing

We use an example to illustrate how KV caches expand the physical design space of an LLM-native data system. Consider a table of building-inspection records, where each tuple contains a textual report and one image:

```
SELECT id, sem_extract(report, "recommended repair action") AS action
FROM Inspections
WHERE sem_filter(photo,  "shows visible water damage")
  AND sem_filter(report, "mentions mold or moisture")
WITH QUALITY precision >= 0.90 AND recall >= 0.90;
```

The query is declarative. It specifies two semantic filters, one over images and one over text, followed by a semantic extraction over the surviving reports. It also specifies an end-to-end quality target on the final output.

Naive physical execution. A straightforward implementation invokes a high-quality model for each semantic operator and each input item. For the image predicate, the model receives the image and the predicate, and produces a Boolean decision. For the text predicate, it receives the report and the predicate. For the extraction operator, it receives the report again and generates the action. Before producing any output, the model must process the input context and construct its internal attention state. For long documents and high-resolution visual inputs, this *prefill* step can dominate the latency of semantic operators.

Cache-aware physical execution. A cache-aware system separates object encoding from query execution. In an offline preprocessing phase, the system feeds each object into the model and materializes its KV cache. For the example above, it may construct caches for every inspection photo and every report. At query time, a semantic operator no longer needs to send the full object through the model from scratch. Instead, it retrieves the corresponding cache, appends an operator-specific suffix that encodes the predicate or extraction request, and performs only the remaining decoding work. The same cached report can be used by the textual filter, by the extraction operator, and by future queries over the same collection, i.e., across multiple semantic operations instead of recomputing it for every call.

Compressed cache profiles. A full KV cache can be large, especially for long contexts and multimodal inputs. Rather than using only the full cache, the system can materialize several cache *profiles* for the same object, each corresponding to a model and a compression ratio. For example, an image may be represented by a full cache, a lightly compressed one, and a more aggressively compressed one. These profiles become distinct physical implementations of the same operator. For the first predicate in the example, the optimizer may determine that a highly compressed image cache is sufficient to reject many obvious non-matches. For the extraction operator, which produces a more information-sensitive output, the optimizer may choose a less compressed cache.

Cascades over cache-aware operators. The physical choices need not be used in isolation. A semantic filter can be implemented as a cascade of increasingly expensive operators. For instance, the image predicate may first use a small model with an aggressively compressed cache. If the model is confident, the tuple is accepted or rejected immediately. If the decision is uncertain, the tuple is forwarded to a less compressed cache profile or to a larger model. This cascade view is useful because not all tuples require the same amount of computation. In the inspection example, many photos may clearly show no water damage, and many reports may clearly not mention moisture. Processing those tuples with the most expensive model and the least compressed cache would waste resources. Conversely, a small model or an aggressive compression profile may be insufficient for borderline cases. A cache-aware cascade gives the optimizer intermediate choices between these extremes.

Speed-up and Optimizer. Compressed KV caches create two complementary speed-up opportunities. First, as the cache has already been computed, execution bypasses the prefill phase and focuses on output generation. This is especially beneficial for semantic filters and short extractions, where the output is small w.r.t the input context. Second, compression reduces the memory footprint of each cached item. Smaller caches allow the executor to fit more items in a GPU batch, improving throughput when evaluating an operator over a collection.

These profiles create intermediate execution choices between the extremes of a cheap approximate operator and an expensive high-fidelity one. Section 3 describes how these choices are profiled and exposed to the optimizer for query execution. Although this example focuses on execution, the same cached representations may also provide useful signals for optimization-time tasks such as estimating

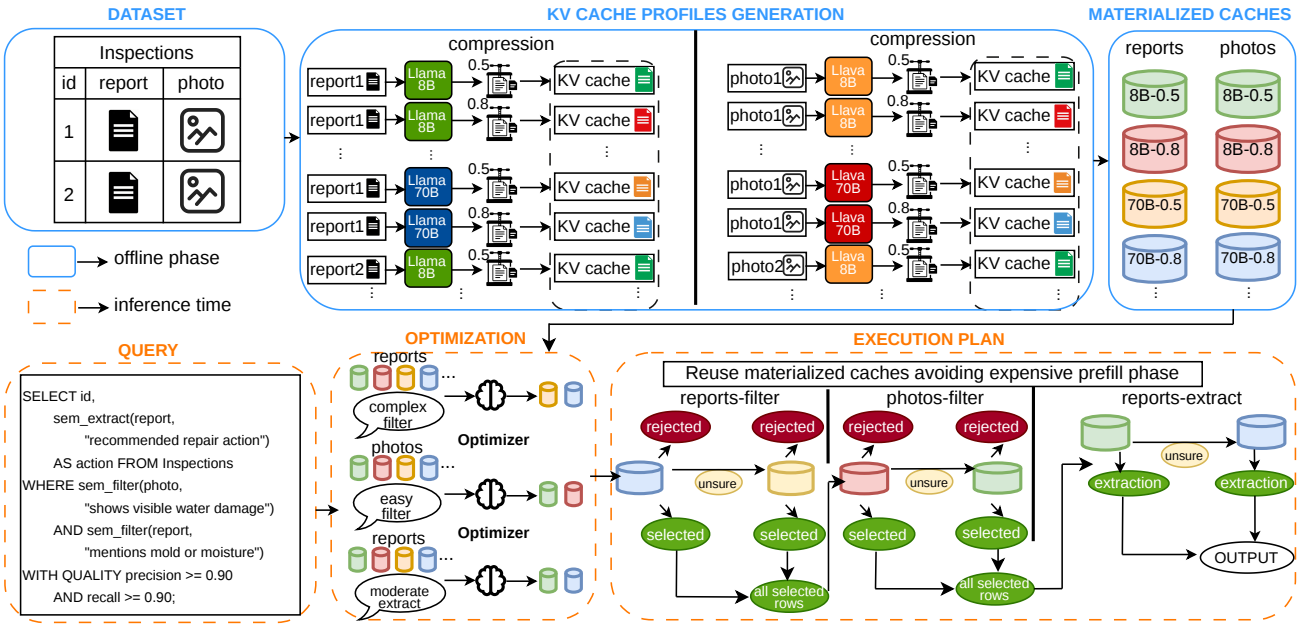

Figure 1: Cache-aware execution of the running example. The user specifies a logical query with semantic filters and an extraction operator. Offline, the system materializes KV-cache profiles for each base object under different models and compression ratios. Online, the optimizer selects a physical implementation for each semantic operator: aggressive compression may be sufficient for easy filtering decisions, while extraction or uncertain cases may use higher-fidelity profiles. The speed-up opportunity comes from bypassing repeated prefill computation and from using smaller compressed caches that permit larger batches and lower latency.

predicate selectivity or operator difficulty; we leave that direction as an opportunity rather than the focus of this article [20].

3 Cache-Aware Physical Design

We now describe how cache-aware physical design is implemented. The design has two phases. Offline, the system profiles candidate physical implementations, including KV-cache-aware choices, and retains a compact set of useful alternatives. Online, the optimizer selects implementations for a logical semantic plan while satisfying user-specified quality targets. We use STRETTO as a reference example, but the pattern is not specific to a particular optimizer or cache-compression method.

3.1 From Semantic Operators to Physical Implementations

A semantic operator specifies a task over data, but not the mechanism used to execute that task, e.g., a filter predicate "shows visible water damage" or "mentions mold or moisture". These operators define the logical meaning of the query. The system can choose among different physical implementations that preserve this meaning up to a requested quality target, e.g., a filter implemented by invoking a large model directly, a smaller model, or a cached representation of the input object.

This separation mirrors the logical–physical distinction in classical database systems. The difference is that, in LLM systems, physical implementations affect not only runtime and resource consumption but also output quality. The optimizer therefore reasons over a multi-dimensional space in which each

implementation has an estimated latency, memory footprint, throughput, and quality.

KV caches expand this space with a new class of physical implementations. The full cache may offer higher quality but require more memory. A compressed cache may be faster and allow larger batches, but may introduce some quality degradation. More precisely, let o be a logical semantic operator, such as a semantic filter or extraction operator. The system associates o with a set of candidate physical implementations:

$$\mathcal{P}(o) = \{p_1, p_2, \ldots, p_k\}.$$

Each implementation p_i specifies how the operator may be executed, for example by choosing a model, a cache profile[1], a compression ratio, a decision threshold, or a cascade policy. We write this configuration as

$$p_i = \langle m_i, c_i, \tau_i \rangle,$$

where m_i is the model, c_i is the cache profile or compression setting, and τ_i denotes operator-specific parameters such as thresholds. The optimizer also needs estimates of how this implementation behaves on the data and query class at hand. These estimates are obtained by profiling p_i on a calibration sample or representative workload:

$$\widehat{\mu}(p_i, o, D) = \langle \widehat{q}_i, \widehat{r}_i \rangle,$$

where $\widehat{q}_i$ is the estimated output quality and $\widehat{r}_i$ is the measured or estimated execution time when applying p_i to operator o over a data sample D. In addition, the system can associate each implementation with a cache footprint s_i, which is determined by the model and compression ratio rather than by the operator output: $s_i = \text{size}(m_i, c_i)$. Thus, quality and runtime are empirical properties of a physical implementation under a given query, data distribution, and hardware, while the cache footprint follows from the materialized profile.

Different operator types may favor different profiles: filters often benefit from aggressive compression, while maps and extractions may require higher-fidelity caches because they generate longer outputs.

3.2 Offline Generation and Pruning of KV-Cache Profiles

The offline phase creates the physical design space exposed to the optimizer. Given a dataset, the system can theoretically materialize KV caches for every base object (e.g., text documents, images) across multiple models and compression ratios. Compression is applied to deliberately trade task fidelity for faster execution and a smaller memory footprint.

However, KV caches are large. Materializing and storing every possible combination of model and compression ratio would incur prohibitive storage costs and bloat the optimizer's search space. Therefore, we must carefully select which profiles to actually materialize.

Constructing the Pareto Frontier Offline. To decide which profiles to keep, we evaluate a grid of candidate model–compression pairs on a calibration sample to measure expected execution time and output quality (e.g., F1 score). The calibration workload need not contain all future queries; it is used to estimate the behavior of each candidate implementation on the class of semantic tasks the system expects to support.

The grid is a design choice: the system starts from a finite set of models and compression ratios chosen by the system designer or from prior profiling. Using these measurements, we construct a *Pareto frontier* offline. We plot the candidates in a runtime–quality space and discard any *dominated* implementations. An implementation is dominated if another profile provides at least the same quality

[1] A cache profile becomes a physical operator only when it is paired with an operator-specific prompt, threshold, or cascade policy.

with a lower runtime. For instance, empirical profiling often reveals that an aggressively compressed 70B model might degrade in quality so much that it matches the accuracy of an uncompressed 8B model, but still runs slower. In such cases, the compressed 70B profile is discarded. The system only physically materializes the KV caches that lie on the non-dominated frontier.

Manual Pruning for Diverse KV-cache Profiles. Moreover, manual pruning of the Pareto frontier can further decrease storage costs and make the optimizer's search space more compact. We favor diversity across the selected points on the Pareto frontier: the final selected profiles should include cheap aggressive-compression options, high-fidelity options, and intermediate choices. Configurations that are too close to already selected points are removed.

This offline pruning yields a compact, pre-computed repository of non-dominated cost–quality trade-offs, reducing storage and optimizer search cost.

3.3 Online Query Planning

At query time, the optimizer's job is greatly simplified. It does not need to reason about raw compression algorithms or search an infinite space of hyperparameters; it is simply handed the compact, pre-computed Pareto frontier of physical operators generated offline.

Given a logical plan containing multiple semantic operators, the optimizer selects one physical implementation from the corresponding frontier for each operator. Because users specify *global* quality targets and errors propagate through the execution pipeline, the optimizer uses the frontier to strategically allocate the query's error budget. It may assign highly compressed, cheap profiles to easy or low-impact operators, while reserving the uncompressed, full-cache implementations for operators that heavily influence final output quality. The offline frontier ensures that the optimizer reasons over implementations that were non-dominated on the calibration workload, rather than over redundant or clearly inferior cache profiles.

3.4 Executing Semantic Operators with Cached Representations

Once the optimizer finalizes the physical plan, the system executes it. For cache-aware operators, the executor loads the selected KV-cache profile from disk instead of re-encoding the raw input object from scratch. The operator simply appends its task-specific suffix (e.g., a natural language filter predicate or an extraction prompt) to the cached state and performs the remaining decoding work.

This approach yields two major runtime benefits:

1. **Bypassing Prefill:** By avoiding the expensive context-encoding phase, execution latency depends mainly on the chosen cache profile and the length of the generated output.

2. **Higher Throughput:** Because offline compression reduces the memory footprint of the surviving cached items, the executor can fit more objects into GPU memory simultaneously. This increases the maximum batch size, significantly improving throughput when evaluating semantic operators over large collections.

4 Illustration in LLM-Native Systems

This section illustrates the effect of cache-aware physical design in LLM-native workloads. The goal is not to introduce a benchmark, but to show two consequences of the design pattern described above. First, KV-cache profiles create a richer runtime–quality space for individual semantic operators. Second,

Table 1: Physical operator variants considered before pruning. Each candidate implementation is defined by an operator type, a model, and a KV-cache compression ratio. Compression ratio 0 denotes the uncompressed KV cache and is also the baseline used in the end-to-end experiment. The optimizer is only exposed to the subset of variants (in bold) retained after Pareto and diversity pruning.

Dataset	Modality	Operators	Candidate Variants 70B	Candidate Variants 8B
MOVIE	Text	filter, map	{0.00, **0.30**, 0.50, **0.60**, **0.80**, 0.90}	{**0.00**, 0.30, **0.50**, **0.80**, 0.90}
ROTOWIRE	Text	filter, map	{0.00, **0.30**, 0.50, **0.60**, **0.80**, 0.90}	{**0.00**, 0.30, **0.50**, **0.80**, 0.90}
ARTWORK	Image	filter, map	{0.00, 0.30, **0.50**, 0.80, **0.90**, **0.99**}	{**0.00**, **0.50**, 0.80, **0.90**, 0.99}

when these profiles are exposed to a query optimizer, the same optimizer can find faster plans under the same quality targets.

We use STRETTO as the reference optimizer in the end-to-end experiment [16]. The comparison is therefore not between two different optimizers, but between two physical design spaces: one where STRETTO can choose among compressed KV-cache profiles, and one where the same optimizer does not have access to these profiles.

4.1 Evaluation Questions

The evaluation focuses on two questions.

Q1: Do KV-cache profiles create useful runtime–quality Pareto frontiers? We first study whether different model–compression configurations produce distinct, non-dominated choices for individual semantic operators.

Q2: Do KV-cache profiles improve end-to-end query execution under fixed quality targets? We then evaluate complete semantic queries with global precision and recall targets. We compare STRETTO with access to the KV-cache frontiers against the same optimizer without compressed physical operators. Both configurations optimize for the same workload and quality targets. The relevant question is whether the richer physical design space allows the optimizer to reduce runtime while preserving the requested guarantees.

4.2 Experimental Setting

We consider semantic workloads over text and image data, following the practice of evaluating semantic query engines across multiple modalities and operator types [7]. For the operator-level frontier experiment, we use three datasets: MOVIE and ROTOWIRE for text operators, and ARTWORK for image operators. These datasets cover different input modalities and different operator behavior, which is useful for showing that cache profiles do not induce a single universal trade-off. Instead, each workload has its own runtime–quality frontier.

For the Pareto frontier study (**Q1**), we evaluate cache-aware implementations obtained from a finite grid of models and compression ratios for each dataset (Table 1). The grid is chosen manually, based on the models available to the system and on a range of compression levels that produce visibly different runtime–quality behavior. The workload used for this study consists of single-operator tasks, i.e., each execution involves eithera semantic filter or map; they characterize the runtime–quality tradeoff of each physical operator. After profiling the grid, we compute the Pareto frontier to prune dominated configurations. To retain a compact set of configurations, we manually further prune, favoring diversity across selected points and removing configurations that are too close together in the runtime-quality trade-off.

Table 2: Number of queries and distribution of operators in the test workload for Q2 (F = Filter, M = Map).

Dataset	F→M	F→F	M→M	F→M→M	F→F→M	F→M→M→M	F→F→M→M	Total
Movie	10	10	10	10	–	10	–	50
Rotowire	10	10	–	10	10	10	10	60
Artwork	10	10	–	10	10	10	10	60

For the end-to-end experiment (**Q2**), we use STRETTO as the optimizer in both configurations [16]. STRETTO is an optimizer that selects physical implementations for semantic operators using profiled runtime and quality estimates, while enforcing global precision and recall targets. The first configuration, STRETTO w/o compression (Uncompressed KV), uses KV caches but only at compression ratio 0, i.e., uncompressed caches. The second configuration, STRETTO w/ compression (Compressed KV), optimizes over the expanded physical space that includes compressed KV-cache profiles. Both configurations benefit from cache reuse and avoid repeated prefill computation; the comparison isolates the additional value of compression as a physical design dimension. They also receive the same logical queries and the same global precision and recall targets. Runtime is the main metric and results are interpreted only among plans that satisfy the requested quality guarantees. In this study, the test workload includes tasks with multiple operators, as shown in Table 2. Starting from a base set of individual semantic filters and joins, two to four semantic operators are randomly combined to construct queries.

4.3 Operator-Level Trade-Offs from Cache Profiles (Q1)

Table 3 shows the runtime–quality behavior of cache-aware physical implementations, averaged over semantic filter and map operators for each dataset. Each result corresponds to one candidate implementation, obtained by selecting a model and a KV-cache compression ratio. The table covers the two text datasets and the image one (ARTWORK).

The table illustrates the main role of compressed KV caches. For a fixed model, increasing compression generally reduces runtime, but may also reduce quality. Across models, larger models tend to provide higher quality, but they are more expensive. The useful points are therefore not concentrated in a single model or a single compression ratio. Instead, different model–compression combinations occupy different regions of the runtime–quality space. For example, a compressed large model can provide an intermediate option between a cheap small model and a full high-quality large model. Conversely, when compression degrades quality too much, a large compressed model may become dominated by a smaller model.

The selected frontier contains diverse points along this curve, rather than every measured configuration. This gives the optimizer a compact set of choices between coarse alternatives such as “small” and “large” models.

4.4 End-to-End Planning with and without KV Caches (Q2)

Figure 2 evaluates the impact of compressed KV-cache profiles on semantic queries. In this experiment, the optimizer is STRETTO in both configurations. The baseline configuration uses uncompressed KV caches, corresponding to compression ratio 0. The cache-aware configuration exposes the same optimizer to the full KV-cache frontier, including compressed profiles. The comparison does not measure the benefit of caching itself (both configurations use cached representations), but it isolates the benefit of adding compressed cache profiles to the physical design space.

Table 3: Runtime–quality trade-off induced by KV-cache profiles. Each row represents a physical implementation defined by a model and compression ratio. Results are averaged over queries (single filter or map operator) for each dataset. Compressed cache profiles add intermediate points between cheap and expensive operators. The Gold Operator is the largest model without compression.

Gold Model: 70B model with no compression **Pareto Optimal**: On the Pareto frontier

		Movie		Rotowire		Artwork	
Model	**Ratio**	**Runtime (s)**	**F1 Score**	**Runtime (s)**	**F1 Score**	**Runtime (s)**	**F1 Score**
70B	0.00	47.59	1.0000	252.86	1.0000	499.51	1.0000
	0.30	33.98	0.9612	166.54	0.9287	360.96	0.9573
	0.50	32.71	0.9228	85.19	0.8833	268.28	0.9361
	0.60	29.83	0.8921	71.75	0.8468	232.38	0.9186
	0.80	24.35	0.7217	51.67	0.7334	123.52	0.8653
	0.90	22.72	0.4656	36.91	0.5858	38.02	0.8479
	0.99	22.93	0.3074	34.31	0.1745	8.89	0.6830
8B	0.00	16.19	0.7161	50.31	0.6285	20.18	0.7410
	0.30	14.70	0.6970	36.82	0.6195	16.64	0.7409
	0.50	12.41	0.6759	27.90	0.6008	12.41	0.7281
	0.80	8.89	0.5852	16.18	0.5038	9.51	0.6783
	0.90	7.12	0.4064	13.56	0.4063	8.29	0.6663
	0.99	–	–	–	–	8.37	0.4971

The comparison is made under the same global precision and recall targets. This point is important: both configurations satisfy the requested statistical quality guarantees. The difference is therefore not that the cache-aware version trades away correctness for speed. Rather, compressed KV-cache profiles give the optimizer additional non-dominated physical implementations, allowing it to find lower-runtime plans that still meet the same quality constraints.

The effect is visible across the datasets in Figure 2. For loose quality targets, the optimizer can often select aggressively compressed profiles or cheap cascade stages, leading to large runtime reductions. For stricter targets, the optimizer uses higher-fidelity profiles more often, but cache-aware planning can still reduce runtime by selecting smaller cache profiles that improve batching and reduce cache-loading costs. The overall message is that KV caches are useful not only as a serving optimization, but as physical operators that improve end-to-end query planning.

The gains are not uniform across datasets. They are largest on Movie, where many predicates are sentiment-style judgments over text. Such tasks remain accurate even under aggressive compression, because the relevant signal is coarse across the review. Stretto can therefore select highly compressed profiles while still satisfying the precision and recall targets. The gains are smaller on Rotowire, where factual details are more sensitive to information loss. This illustrates why exposing a frontier is useful: the optimizer can exploit compression when the task is robust to it and fall back to higher-fidelity profiles otherwise. Finally, for Artwork, the KV caches are very large, so avoiding the loading of multiple cache profiles can outweigh the benefit of using smaller compressed caches in this small-scale experiment. However, this should not be interpreted as evidence that uncompressed KV caches are preferable in general: for realistic corpora with thousands of images, the uncompressed cache repository may become too large to store, load and batch, making compression necessary to keep the physical design space practical.

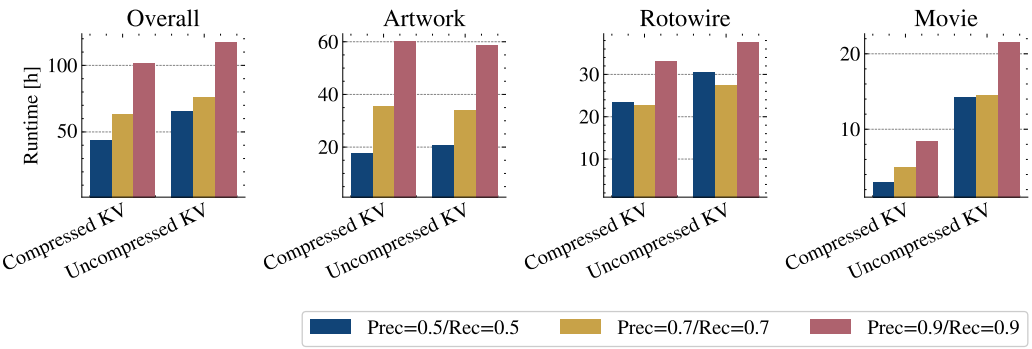

Figure 2: Runtime comparison of STRETTO with uncompressed KV caches and STRETTO with compressed KV-cache profiles. Both configurations use the same optimizer and workload, reuse KV caches, and satisfy the requested statistical guarantees on output quality. The cache-aware configuration exposes a richer physical design space by adding compressed profiles, enabling faster plans under the same precision and recall targets.

Table 4: KV-cache storage sizes (in GB) across textual and visual datasets for different combinations of model and compression ratio.

8B Model				70B Model			
Ratio	**Movie**	**Rotowire**	**Artwork**	**Ratio**	**Movie**	**Rotowire**	**Artwork**
0.0	6.8	38	298	0.0	17	93	5800
0.5	3.4	19	149	0.3	12	65	4060
0.8	1.4	7.4	60	0.8	3.3	19	1160

4.5 Memory Requirements and Compression Impact

To quantify the storage requirements of maintaining multiple KV-cache configurations, Table 4 reports the memory footprints across our benchmark workloads. The numbers show why cache profiles cannot be materialized indiscriminately: for example, the uncompressed 70B cache footprint reaches 93GB on ROTOWIRE and 5800GB on ARTWORK, while compression substantially reduces these requirements. In the end-to-end experiments, exposing compressed profiles to the optimizer yields average speedups for all precision/recall targets w.r.t. a baseline that uses uncompressed KV caches.

4.6 Takeaways

The experiments support two conclusions. First, compressed KV-cache profiles create a richer Pareto frontier of physical operator implementations, with useful intermediate choices between cheap low-quality operators and expensive high-quality ones. Second, exposing this frontier to STRETTO improves end-to-end execution: the optimizer can select faster physical plans while satisfying the same statistical quality guarantees.

5 Conclusion and Outlook

LLM-native data systems need a richer physical layer than model and prompt selection alone. As semantic operators become part of declarative query engines, the system must decide not only *which* model to invoke, but also *how* inference state should be materialized, compressed, reused, and exposed

to the optimizer. KV caches provide one concrete example of this broader principle. When treated as physical design structures, they create additional implementations of semantic operators and fill gaps in the runtime–quality space. Pareto frontiers then make this expanded space manageable: the optimizer can reason over a compact set of non-dominated alternatives instead of over a large collection of redundant model–compression choices.

Empirical results illustrate the potential of this design. At the operator level, compressed KV-cache profiles create diverse runtime–quality trade-offs across workloads. At the query level, exposing these profiles to an optimizer enables faster physical plans while preserving the same quality guarantees.

Several research directions remain open.

Cache-aware physical design. Current cache-profile selection relies on a finite grid of models and compression ratios. Future systems should make this process more adaptive. Given a workload, hardware budget, and quality target, the system should decide which profiles to materialize, which to discard, and when to refresh them. More generally, memory should become an explicit constraint in semantic query optimization, rather than an implicit property of the serving backend. An optimizer would then need to choose physical operators subject not only to runtime and quality constraints, but also to storage and GPU-memory budgets for cached representations. This raises questions similar to index and materialized-view selection, but with an additional quality dimension. Existing semantic query optimizers would need to expose memory footprints in their cost models, propagate memory requirements across multi-operator plans, and coordinate with the serving layer on batching, cache eviction, and recomputation policies.

Robust quality estimation. The optimizer depends on estimates of the quality and runtime of each physical implementation. These estimates are currently obtained from calibration workloads, but future queries may differ from the calibration sample. A key challenge is to make quality estimates robust under workload shift, model updates, and changing data distributions. This is especially important when the system promises end-to-end quality guarantees.

Cache-derived statistics. Although this article focuses on execution, KV caches may also support optimization-time statistics. Cached representations can help estimate predicate selectivity [20], join survival rates, or the difficulty of an operator. This direction would make caches play a role closer to classical database statistics: useful not only for faster access, but also for better planning. However, it requires careful validation, since cache-derived signals may be model-specific and workload-dependent.

Beyond KV caches. KV caches are one instance of a more general idea: inference artifacts can become physical design structures. Future LLM-native systems may expose other reusable internal representations, intermediate reasoning traces, retrieval states, or learned operator summaries to the optimizer. As the community shifts toward agentic workflows and AI-powered data clouds [1, 2], caching and reusing the intermediate tool-use and reasoning traces of these agents will become a pivotal dimension of physical design. The broader research question is how to define a principled physical algebra for LLM-native execution, where inference state, quality estimates, memory footprint, and runtime all become first-class optimization dimensions.

Acknowledgments

This work was funded by the DFG/ANR Project MAgiQ (ANR24-CE92-0077; DFG, German Research Foundation – Project No. 545611510), the LOEWE Spitzenprofessur programme (III 5-

519/05.00.003(0005)), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC3057/1 "Reasonable Artificial Intelligence", Project No. 533677015), and by the French government, through the 3IA Côte d'Azur Investments in the IA-cluster project managed by the National Research Agency (ANR-23-IACL-0001). This project was provided with resources by GENCI at IDRIS, thanks to grants 2025-AD010616649 and 2025-AD010616180. We also thank DFKI and hessian.AI.

References

[1] Elaine Ang, Chenxi Huang, Georgios Liargkovas, Jerry Liu, Jinhui Liu, Nikos Pagonas, Charlie Summers, Haonan Wang, Jiakai Xu, Tianle Zhou, Yusen Zhang, Zhou Yu, Zhuo Zhang, Tianyi Peng, Kostis Kaffes, and Eugene Wu. Agentic data environments. *IEEE Data Eng. Bull.*, 50(1):5–21, 2026.

[2] Yeounoh Chung, Thibaud Hottelier, Cosmin Arad, Brenton Milne, Per Jacobsson, Sam Idicula, Fatma Özcan, Alon Y. Halevy, and Yannis Papakonstantinou. Architecting the ai-powered agentic data cloud. *IEEE Data Eng. Bull.*, 50(1):22–39, 2026.

[3] Giulio Corallo and Paolo Papotti. FINCH: prompt-guided key-value cache compression for large language models. *Trans. Assoc. Comput. Linguistics*, 12:1517–1532, 2024.

[4] Alessio Devoto, Maximilian Jeblick, and Simon Jégou. Expected attention: Kv cache compression by estimating attention from future queries distribution, 2025.

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

[6] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

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

[8] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[9] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, 2025.

[10] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. Optimizing LLM queries in relational data analytics workloads. In *Eighth Conference on Machine Learning and Systems*, 2025.

[11] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Representation Learning*, volume 2025, pages 34433–34448, 2025.

[12] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators and their optimization: Enabling llm-based data processing with accuracy guarantees in lotus. *Proc. VLDB Endow.*, 18(11):4171–4184, September 2025.

[13] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Karthik Kefato, Tate Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems (MLSys)*, 5, 2023.

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

[15] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael J. Cafarella. Abacus: A cost-based optimizer for semantic operator systems. *CoRR*, abs/2505.14661, 2025.

[16] Gabriele Sanmartino, Matthias Urban, Paolo Papotti, and Carsten Binnig. The stretto execution engine for llm-augmented data systems, 2026.

[17] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. Logical and physical optimizations for sql query execution over large language models. *Proc. ACM Manag. Data*, 3(3), 6 2025.

[18] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *Proc. VLDB Endow.*, 18(9):3035–3048, September 2025.

[19] Matthias Urban and Carsten Binnig. CAESURA: language models as multi-modal query planners. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024.

[20] Matthias Urban, Vu Huy Nguyen, Gabriele Sanmartino, Paolo Papotti, and Carsten Binnig. Selectivity estimation for semantic filters on image data, 2026.

[21] Lindsey Linxi Wei, Shreya Shankar, Sepanta Zeighami, Yeounoh Chung, Fatma Ozcan, and Aditya G. Parameswaran. Multi-objective agentic rewrites for unstructured data processing, 2026.

[22] Sunny Yasser, Anas Dorbani, and Amine Mhedhbi. Factorized and vectorized execution: Optimizing analytical and semantic queries over relations. *Proc. ACM Manag. Data*, 4(3), 2026.

[23] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. Cut costs, not accuracy: Llm-powered data processing with guarantees, 2025.