Optimal Block Nested Loops Implementations for Semantic Joins

Immanuel Trummer
Cornell University
itrummer@cornell.edu

Abstract

Semantic query processing engines often support semantic joins, enabling users to match rows that satisfy conditions specified in natural language. Such join conditions can be evaluated using large language models (LLMs) that solve novel tasks without task-specific training.

Currently, many semantic query processing engines implement semantic joins via nested loops, invoking the LLM to evaluate the join condition on row pairs. Instead, this paper proposes a novel algorithm, inspired by the block nested loops join operator implementation in traditional database systems. The proposed algorithm integrates batches of rows from both input tables into a single prompt. The goal of the LLM invocation is to identify all matching row pairs in the current input. The paper introduces formulas that can be used to optimize the size of the row batches, taking into account constraints on the size of the LLM context window (limiting both input and output size). A formal analysis of asymptotic processing costs, as well as empirical results, demonstrates that the proposed approach reduces costs significantly and performs well compared to join implementations used by recent semantic query processing engines.

1 Introduction

Several recent systems [2, 4, 11, 18, 23, 26–28] expand SQL by introducing semantic operators. Those operators, including, for instance, semantic filters and semantic sort operators, are configured via natural language instructions and evaluated by large language models (LLMs). Compared to traditional relational operators, the per-byte processing overheads of such operators are typically higher by many orders of magnitude. This means, in the context of semantic queries, processing overheads are typically dominated by overheads due to semantic operators. This makes it crucial to make those operators as efficient as possible.

This paper focuses on a semantic version of a classical relational operator: the relational join. Semantic joins, as defined by systems like LOTUS [22], enable users to describe the join condition in natural language. Note that this paper does not explicitly focus on equality joins. Instead, it focuses on general theta-joins [17] with natural language predicates. For instance, such join operators are useful in the following example scenario.

Example 1: To investigate a large corporation, prosecutors plan to analyze a large collection of emails. The goal is to compare emails to statements made by executives of that company. Of particular interest are instances where an email contradicts statements made by the defendants. This can be modeled as a join between two data sets, one containing statements and the other one containing emails. The join predicate can be formulated in natural language as "pairs of documents that contradict each other."

A naive implementation of the semantic join operator, implemented in several semantic query processing engines at the time of writing, uses the LLM to evaluate the join condition on each pair of input tuples (i.e., a tuple nested loops join). The amount of data sent to the LLM is directly proportional to the product of the cardinalities of both input tables. Similarly, the amount of data generated by the LLM (one token per pair of input tuples with the predicate evaluation result) is proportional to the cardinality product. LLM providers such as OpenAI and Anthropic calculate processing fees as a weighted sum over the number of input and output tokens. Hence, monetary processing fees are proportional to the cardinality product as well.

The aforementioned approach uses the LLM like a user-defined function predicate. However, this approach does not exploit the flexibility of LLMs, enabling them to follow almost arbitrary instructions in natural language. Instead of merely returning a Boolean result for each tuple pair, we can instruct the LLM to find all matching pairs, according to the join predicate, when given not two tuples but two larger batches of tuples in the input.

We can exploit the flexibility of LLMs for a block nested loops implementation of the semantic join operator. In each call to the LLM, we send tuple batches from both input tables and retrieve all matching pairs in the LLM reply. Clearly, this approach reduces the number of LLM calls, as the number of pairs of tuple batches is smaller than the number of tuple pairs (assuming that batches contain at least two tuples). However, the amount of data read and generated in each LLM call is higher, raising the question of whether this approach leads to cost benefits. Intuitively, the amount of data read and generated per LLM call grows only linearly as a function of the tuple batch size. On the other hand, the number of LLM calls decreases quadratically as a function of the tuple batch size, leading to an asymptotic advantage. Section 3 analyzes time complexity in more detail.

Ideally, we would like to send all tuples from both input tables to the LLM in a single invocation, enabling us to retrieve all matching rows in a single invocation. In that case, the number of tokens read and generated (and therefore monetary execution fees) are proportional to the amount of input and output data, and therefore optimal. However, this approach is not feasible in general since the LLM context window, the maximal number of tokens that can be read and generated in a single invocation, is bounded. The bounded context size raises the following research question: *How to choose an optimal batch size for the two input tables to minimize processing costs while complying with the size bounds?*

The constraint on context window size relates to constraints imposed by limited main memory in traditional block nested loops join algorithms. Both (limits on context size and limits on main memory size) imply lower bounds on the number of blocks to compare. For traditional block nested loops join, the optimal allocation strategy uses as much main memory as possible to store large blocks from one of the two input relations while reserving a minimal amount of main memory for the output and the second relation. However, the equivalent allocation strategy is sub-optimal when applied to the context window, as demonstrated next.

Example 2: We join two tables R and S, each containing 100 tuples with equal size. The context window can fit 20 tuples, and the join result is empty. Using batches of only one tuple for R and a maximal batch size of 19 for S leads to $\lceil (100/1) \cdot (100/19) \rceil = 527$ LLM invocations (each incurs the cost of reading 20 tuples). On the other hand, using equal-sized batches with ten tuples for both tables leads to only $\lceil (100/10) \cdot (100/10) \rceil = 100$ LLM invocations (with the same per-invocation cost as before).

The example illustrates that the batch size matters and that the allocation strategy used by the traditional block nested loops join cannot translate to the semantic join version. The example is simplistic as it assumes that tuples from input tables have the same size. As shown later, the tuple size does influence the optimal batch allocation strategy. Furthermore, the example focuses on a special case in which the join result is empty. In general, as the context size bound restricts the sum of input and

Algorithm 1 Block nested loops join algorithm for semantic joins, executed via large language models.

```
// Perform block join between relations R_1 and R_2
// with join condition j and using block sizes b_1 and b_2.
function BlockJoin(R_1, R_2, j, b_1, b_2)
    // Initialize result set
    R ← ∅
    // Partition input into batches
    B_1 ← {B_i ⊆ R_1 | R_1 = ⊍_i B_i, ∀i | B_i| = b_1}
    B_2 ← {B_i ⊆ R_2 | R_2 = ⊍_i B_i, ∀i | B_i| = b_2}
    // Iterate over pairs of batches
    for B_1 ∈ B_1 do
        for B_2 ∈ B_2 do
            // Create prompt for LLM
            P ← BlockPrompt(B_1, B_2, j)
            // Get raw answer from LLM
            A ← InvokeLLM(P)
            // Check for overflow
            if A[−1] ≠ Finished then
                return Overflow
            end if
            // Extract result tuples
            R ← R ∪ ExtractTuples(B_1, B_2, A)
        end for
    end for
    // Return join result
    return R
end function
```

output tuples, the allocation strategy must allocate sufficient space in the context window to output matching tuples. The primary technical contribution in this paper is a provably optimal token allocation strategy that chooses optimal sizes for the input batches, taking into account all of the aforementioned factors.

The remainder of this paper is organized as follows. Section 2 introduces the semantic block nested loops join, exploiting LLMs to identify matching tuples across two tuple batches. Section 3 shows how to optimize batch sizes for that join operator if the selectivity of the join predicate is known. Section 4 reports on experiments, comparing all join operators in different scenarios and according to different metrics. Finally, Section 5 contrasts the work presented in this paper with prior work.

2 Block Nested Loops Join

This section introduces a variant of the block nested loops join, as well as an associated cost model.

2.1 Algorithm

Algorithm 1 takes as input two tables (R_1 and R_2) and a join condition j. In addition, Algorithm 1 uses input parameters b_1 and b_2, representing the number of tuples from the first and second table that are

```
Find indexes x,y where x is the number of an entry
in collection 1 and y the number of an entry in
collection 2 such that [j] (make sure to catch
all pairs!)!
Separate index pairs by semicolons.
Write "Finished" after the last pair!
Text Collection 1:
1. [B1[1]]
2. [B1[2]]
...
Text Collection 2:
1. [B2[1]]
2. [B2[2]]
...
Index pairs:
```

Figure 1: Prompt template used for block nested loops join (instantiated by Function BlockPrompt in pseudo-code).

processed together as one batch. The choice of those parameter values is non-trivial and analyzed in the following section.

Algorithm 1 starts by partitioning tuples from both input tables, using the specified batch sizes (the pseudo-code is slightly simplified, based on the assumption that the number of tuples in each table is a multiple of the batch sizes). Instead of iterating over pairs of tuples, the algorithm iterates over pairs of tuple batches. For each pair of batches, the algorithm uses a language model to retrieve all tuple pairs that satisfy the join condition. Instead of invoking the language model for each tuple pair, Algorithm 1 invokes the model only once for each pair of tuple batches.

Figure 1 shows the corresponding prompt template, instantiated by Function BlockPrompt. The prompt contains placeholders for the join condition, `[j]`, and for the tuples in each block, denoted as `[Bi[j]]` where `i` is the index of the table containing the tuples and `j` the index of a tuple within the current tuple batch. The template starts with instructions, directing the language model to find pairs of indexes that represent matching tuples. Each pair of matching tuples is denoted as `x,y` where `x` refers to the position of a tuple from the first batch and `y` to the position of the tuple within the second batch. While seemingly redundant, the additional instructions `make sure to catch all pairs!` are important to encourage the language model to generate a complete result. The number of matching tuple pairs may range from zero to the product of the two input batch sizes. The prompt instructs the language model to use semicolons to separate different index pairs.

The number of output tokens is limited, determined by the properties of the used language model. If reaching the limit in terms of output tokens, the answer generated by the language model becomes inconclusive. It is unclear whether the language model found all matching pairs or ran out of tokens before being able to generate complete output. For that reason, the prompt in Figure 1 instructs the language model to mark the last matching index pair with the word "Finished". If the word "Finished" concludes the output, even when reaching the token limit, it is clear that the output contains all matching tuples (at least all matches that the language model is able to find). Finally, the prompt template contains tuples from the two input batches, each prefixed by a batch-specific index number.

In principle, asking the language model to write complete result tuples (i.e., to copy matching input

tuples) is possible as well. However, as the cost for generating output is proportional to the number of generated tokens (and, at least for some models, generating tokens is more expensive than reading tokens), generating index pairs, rather than result tuples, reduces processing fees.

Algorithm 1 sends prompts generated for the current pair of batches to the language model to retrieve an answer. First of all, Algorithm 1 checks whether a complete result (according to the capabilities of the language model) was generated. As the prompt instructs the language model to terminate output with the keyword "Finished", the algorithm checks the last word in the answer using the (Python-inspired) notation $A[-1]$. If the keyword is not "Finished", the join operator returns the flag **Overflow**. This means that the result is incomplete, and the settings for the batch sizes, b_1 and b_2, are invalid. This can happen if initial estimates on the selectivity of the join condition, determining the number of output tokens that are generated, turn out to be erroneous. If so, the batch sizes can be recalculated based on a less optimistic (i.e., higher) selectivity estimate. In the experiments, we show that a simple adaptive strategy, multiplying an initial (optimistic) selectivity estimate with a constant factor α, whenever an overflow is encoutered can deal effectively with incorrect selectivity estimates.

Function ExtractTuples (the pseudo-code is omitted due to space restrictions) translates index pairs in the answer into tuple pairs.

2.2 Cost Model

Parameters r_1 and r_2 denote the number of rows in the first and second table respectively. Parameters s_1, s_2, and s_3 denote the (token) size of tuples in the two input tables and per result index pair (s_3), respectively. Parameter p is the size of the tuple-independent parts of the prompt represented in Figure 1 (i.e., all text except for the parts that describe the input tuples). Parameter σ represents the selectivity of that join condition, i.e., the ratio of input tuple combinations satisfying the join condition. Finally, parameter g represents the relative cost of generating tokens, relative to the cost of reading tokens. For some LLMs, the cost of reading and generating tokens is equal (i.e., $g = 1$) but for some of the more recent models (e.g., GPT-4), the cost of generating tokens is higher than the cost of reading them (i.e., $g > 1$). Parameters b_1 and b_2 denote the batch sizes for the first and second table (i.e., the input parameters in Algorithm 1). Parameters related to size and selectivity (namely, parameters s_1, s_2, s_3, r_1, r_2, and σ) depend on data properties whereas g depends on the LLM and p is specified by the user. Only the values for parameters b_1 and b_2 can be chosen.

The following lemmata and theorems calculate the number of LLM invocations, the number of tokens processed per invocation, and the cost per LLM invocation. Note that the following analysis is simplifying as it treats all parameters as continuous (e.g., r_1/b_1, as opposed to $\lceil r_1/b_1 \rceil$, when calculating the number of batches for the first table). This facilitates the analysis in the following sections, applying differentiation to obtain optimal values for tuning parameters b_1 and b_2.

Lemma 1: The number of tokens processed per LLM invocation is given by $p+b_1 \cdot s_1+b_2 \cdot s_2+b_1 \cdot b_2 \cdot \sigma \cdot s_3$.

Proof: Each prompt contains a batch of b_1 tuples from the first table with a size per tuple of s_1, i.e., $b_1 \cdot s_1$ is the number of tokens used to represent entries from the first table. Similarly, entries from the second table consume $b_2 \cdot s_2$ tokens. The expected number of join result tuples is given by $b_1 \cdot b_2 \cdot \sigma$ and their size by $b_1 \cdot b_2 \cdot \sigma \cdot s_3$. Finally taking into account tokens required for the join task description (p) yields the postulated size formula.

Lemma 2: The cost per LLM invocation is given by the formula $p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g$.

Proof: The proof is similar to the one of Lemma 1. Costs are proportional to the number of tokens, except that it distinguishes tokens read from generated tokens. The LLM only generates tokens associated

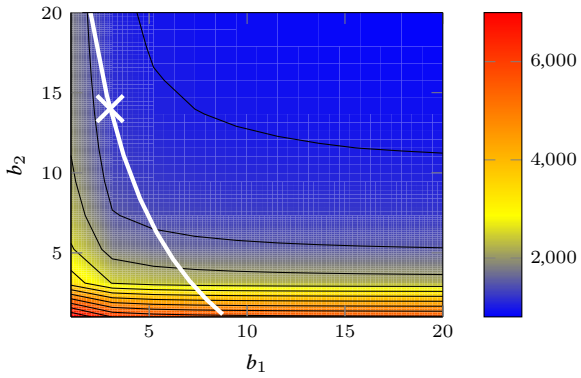

Figure 2: Illustrating join processing costs as a function of the two input batch sizes (b_1 and b_2), using $r_1 = 50$, $r_2 = 10$, $s_1 = 10$, $s_2 = 2$, $s_3 = 1$, $\sigma = 1$, $g = 1$, $p = 1$. All solutions under the white curve use prompts with a size at or below 100 tokens. The white X marks the solution with minimal cost among all solutions with a prompt size of up to 100 tokens.

with the join result. Therefore, the number of corresponding tokens ($b_1 \cdot b_2 \cdot \sigma \cdot s_3$) is scaled by factor g to obtain associated costs.

Lemma 3: The number of LLM invocations for join processing is given by the formula $(r_1/b_1) \cdot (r_2/b_2)$.

Proof: This follows from the definition of Algorithm 1. The LLM is called in each iteration of the inner-most loop. The outer loop iterates r_1/b_1 times whereas the inner loop iterates r_2/b_2 times.

Corollary 1: Total join processing costs are given by the formula $c(b_1, b_2) = (r_1/b_1) \cdot (r_2/b_2) \cdot (p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$.

Proof: This is a direct consequence of Lemmas 2 and 3, obtained by multiplying the cost per LLM invocation with the number of LLM invocations.

3 Optimizing Batch Sizes

Processing fees of the block join, introduced in the previous section, depend on settings for the batch sizes (parameters b_1 and b_2). This section shows how to optimize batch sizes as a function of the input properties. The following example illustrates how processing fees depend on the batch size.

Example 3: Figure 2 plots join cost for an example scenario. A-priori, choosing higher values for b_1 and b_2 seems preferable. However, in practice, the values of b_1 and b_2 are bounded by limits imposed by the LLM on the number of tokens read and generated per invocation. The white line in Figure 2 marks value combinations for b_1 and b_2 for which the number of processed tokens reaches 100. Given a limit on processed tokens, we want to find values for b_1 and b_2 that comply with that token limit (in Figure 2, those are the points below the white line) while minimizing costs under that constraint. The white X marks the optimal solution in Figure 2.

3.1 Analyzing Costs

The combined input and output size per LLM invocation is generally limited, either by a hard bound representing the maximal input and output size that a model can accept or by a (smaller) bound, representing the maximal size for which the model is deemed accurate enough. The second bound is motivated by the observation that LLMs tend to become less reliable with growing input sizes. In the following, t denotes the maximal number of tokens that can be used per LLM invocation. To simplify the following formulas, t does not take into account the size of the task description, p, which remains static over all prompts. In other words, t is obtained by already subtracting p from the LLM-specific size bound. To comply with the size limit, the following equation must hold.

$$b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma \leq t \tag{2}$$

This raises the question of whether or not choosing values for b_1 and b_2 that lead to LLM invocations using less than the maximally allowed number of tokens is efficient. The following theorem shows that this is not the case.

Theorem 3.1: Maximizing the number of tokens processed per LLM invocation minimizes processing costs.

Proof: Assume that the prompt size is below the threshold, i.e., $b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma < t$. Furthtermore, without restriction of generality, assume that b_1 can be replaced by $b_1^* = \alpha \cdot b_1$ for an $\alpha > 1$ such that $b_1^* \cdot s_1 + b_2 \cdot s_2 + b_1^* \cdot b_2 \cdot s_3 \cdot \sigma \leq t$. How do total processing costs with b_1 ($c(b_1, b_2)$) relate to the ones with b_1^* ($c(b_1^*, b_2)$)? It is $c(b_1^*, b_2) = (r_1/b_1^*) \cdot (r_2/b_2) \cdot (p + b_1^* \cdot s_1 + b_2 \cdot s_2 + b_1^* \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$. This can be rewritten as $(r_1/(b_1 \cdot \alpha)) \cdot (r_2/b_2) \cdot (p + b_1 \cdot \alpha \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot \alpha \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$, which simplifies to $(r_1/b_1) \cdot (r_2/b_2) \cdot (p/\alpha + b_1 \cdot s_1 + b_2 \cdot s_2/\alpha + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g)$. Since $\alpha > 1$, it is $c(b_1^*, b_2) \leq (r_1/b_1) \cdot (r_2/b_2) \cdot (p + b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot \sigma \cdot s_3 \cdot g) = c(b_1, b_2)$. If replacing b_2 with $b_2 \cdot \alpha$ with $\alpha > 1$, similar reasoning shows that the cost can only decrease. Hence, increasing the number of tokens processed per LLM invocation, if possible, decreases costs.

Example 4: Consider the cost function depicted in Figure 2. As discussed before, the white curve marks points at which the number of tokens processed per LLM invocation equals the threshold. Due to Theorem 3.1, values for b_1 and b_2 that minimize join processing costs while complying with token limits must be on that curve.

The following lemma shows that the optimal value for b_2 can be expressed as a function of b_1 (denoted as the function $b_2(b_1)$).

Lemma 4: Any solution minimizing $c(b_1, b_2)$ satisfies the equation $b_2 = b_2(b_1) = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$.

Proof: Due to Theorem 3.1, setting $b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma = t$ minimizes processing costs. This equation can be rewritten to $b_2 \cdot (s_2 + b_1 \cdot s_3 \cdot \sigma) = t - b_1 \cdot s_1$. Therefore, the optimal value for b_2 is given as $b_2 = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$

According to Lemma 4, substituting each occurrence of b_2 in the join cost function with $b_2(b_1)$ yields minimal processing costs:

$$
\begin{aligned}
& c^*(b_1) := c(b_1, b_2(b_1)) \\
=& \frac{r_1 \cdot r_2}{b_1 \cdot b_2(b_1)} \cdot (p + b_1 \cdot s_1 + b_2(b_1) \cdot s_2 + b_1 \cdot b_2(b_1) \cdot s_3 \cdot \sigma \cdot g) \\
=& \frac{r_1}{b_1} \cdot r_2 \cdot (\frac{p + b_1 \cdot s_1}{b_2(b_1)} + s_2 + b_1 \cdot s_3 \cdot \sigma \cdot g) \\
=& \frac{r_1}{b_1} \cdot r_2 \cdot (\frac{p + b_1 \cdot s_1}{(t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)} + s_2 + b_1 \cdot s_3 \cdot \sigma \cdot g) \\
=& r_1 \cdot r_2 \cdot (\frac{(s_2/b_1 + s_3 \cdot \sigma) \cdot (p + b_1 \cdot s_1)}{(t - b_1 \cdot s_1)} + \frac{s_2}{b_1} + s_3 \cdot \sigma \cdot g)
\end{aligned}
$$

Hence, the problem of minimizing a function with two parameters ($c(b_1, b_2)$) under constraints reduces to the problem of minimizing a function that depends on a single parameter ($c^*(b_1)$).

3.2 Optimizing Costs

We minimize join processing costs, i.e., $c^*(b_1)$, by a suitable choice for b_1. This means we are searching for minima of $c^*(b_1)$. For b_1^* to be a minimum of $c^*(b_1)$, the following conditions must hold:

$$
c^* b_1(b_1^*) = 0 \qquad [2] c^* b_1(b_1^*) > 0
$$

The first-order derivative of c^* is given as follows:

$$
c^* b_1 = r_1 r_2 (t + p)[\frac{b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t}{(t - b_1 s_1)^2 b_1^2}] \tag{3}
$$

Lemma 5: For c^*, $b_* = [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$ is a critical point (i.e., the first-order derivative is zero).

Proof: It is $r_1 r_2 (t + p) > 0$ since all involved terms are positive. Similarly, it is $(t - b_1 s_1)^2 b_1^2 > 0$. Therefore, the derivative of c^* reaches zero iff $b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t = 0$. This is a quadratic equation in b_1. The roots are therefore given by $(-2 s_1 s_2 \pm \sqrt{(2 s_1 s_2)^2 - 4 (s_1 s_3 \sigma)(-s_2 t)})/(2 s_1 s_3 \sigma)$ which simplifies to $[-s_1 s_2 \pm \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$. Also, as b_1 represents the batch size, it must be positive. Hence, the only valid solution is $[-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$. Note that this solution is guaranteed to be positive since $s_1 s_2 < \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}$.

Theorem 3.2: For c^*, $b_* := [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma)$ is a minimum.

Proof: The theorem holds if $d^2 c^*/d b_1^2 > 0$ at b_* since b_* is a critical point, according to Lemma 5. Set $u(b_1) = b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t$ and $v(b_1) = (t - b_1 s_1)^2 b_1^2$. The first-order derivative of c^*, dc^*/db_1, is $r_1 r_2 (t+p) u(b_1)/v(b_1)$, according to Equation 3. Due to the quotient rule, it is $d^2 c^*/d b_1^2 = r_1 r_2 (t+p)[u'v - uv']/v^2$ where $u' = du/db_1$ and $v' = dv/db_1$. As outlined in the proof of Lemma 5, $u(b_*) = 0$. Hence, at b_*, the second-order derivative $d^2 c^*/d b_1^2$ simplifies to $r_1 r_2 (t + p)[u'v]/v^2$. It is $u' = d/db_1 [b_1^2 s_1 s_3 \sigma + b_1 2 s_1 s_2 - s_2 t] = 2 b_1 s_1 s_3 \sigma + 2 s_1 s_2$. As all constants appearing in this equation are positive with $s_1 > 0$ and $s_2 > 0$, u' is strictly positive for positive values of b_1. Note that $b_1 s_1 < t$ since the token threshold t is at least equal to the number of tokens used for representing tuples from the first and second table, $b_1 s_1 + b_2 s_2$, with $b_2 s_2 > 0$ (since each prompt must contain non-empty input from both tables to be useful). Therefore, v is strictly positive for all values of b_1. This implies that $d^2 c^*/d b_1^2$ is greater than zero at b_*.

Example 5: In the example depicted in Figure 2, we have $s_1 = 10$, $s_2 = 2$, $\sigma = s_3 = 1$. Therefore, it is

$$\begin{aligned} b_* =& [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}]/(s_1 s_3 \sigma) \\ =& [-10 \cdot 2 + \sqrt{10^2 \cdot 2^2 + 10 \cdot 2 \cdot 1 \cdot 1 \cdot 100}]/(10 \cdot 1 \cdot 1) \\ =& [-20 + \sqrt{2400}]/10 \approx 3 \end{aligned}$$

This means selecting batches of three tuples from the first table is optimal (i.e., setting $b_1 = b_* \approx 3$). According to Lemma 4, the optimal number of tuples per batch for the second table is determined as $b_2 = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$ and, for $b_1 = 3$, it is $b_2 = (100 - 3 \cdot 10)/(2 + 3 \cdot 1 \cdot 1) = 14$. Hence, setting $b_1 = 3$ and $b_2 = 14$ minimizes cost under the per-prompt token limit. In Figure 2, the white X marks that point.

4 Experimental Results

The following experiments evaluate the join operators. Section 4.1 describes the experimental setup. Section 4.2 reports on the experimental results.

4.1 Experimental Setup

The experiments use OpenAI's GPT-4.1 Mini model (GPT-4.1-mini-2025-04-14, priced at 40 cents per million input tokens and USD 1.6 per million output tokens). Join operators are implemented in Python 3.11, using OpenAI's Python client in version 1.12. GPT is invoked with a per-request timeout of 300 seconds. The temperature parameter of GPT is set to zero, thereby minimizing randomness in output generation. For the block join, the "Finished" token, marking the end of a complete join result, is used in the stopping condition for output generation (parameter "stop"). We also experiment with an adaptive version of the block nested loops join, starting with a selectivity estimate of $\sigma = 0.001$ that is increased by a factor of $\alpha = 4$, whenever an overflow occurs (the batch sizes are recalculated). Unless noted otherwise, GPT-4.1 is used with a maximal context size of 4,000 tokens. The experiments also evaluate a baseline algorithm ("embedding join"), using OpenAI's text-embedding-3-small model to calculate embedding vectors for each of the tuples in the input tables. Then, each tuple is matched to the tuple with the most similar embedding vector from the other table (based on cosine similarity). Furthermore, the experiments evaluate LOTUS 1.1.4 [22], using the default implementation of the semantic join operator. All experiments are executed on an Apple M1 MacBook Air laptop with 16 GB of RAM, using macOS Sonoma 14.2.1.

The experiments consider six scenarios, some of which connect to the use cases discussed in the introduction. The "Email" scenario, loosely based on the investigation surrounding the Enron scandal, uses language models to find inconsistencies between statements made by defendants and the content of email messages, exchanged by them and their co-workers. It joins one table containing statements of the form "[Name]: I first heard about the losses in February 2022" with a larger table containing short emails of the form "I first told [Name] about the losses [TimeFrame]". Here, [Name] is one of ten common names and [TimeFrame] is a specification of a time frame that either complies, or contradicts the statement by the corresponding defendant. The scenario uses the join condition "the two texts contradict each other." The second scenario ("Reviews") is based on the IMDB movie reviews, available for instance on Kaggle[1]. The goal is to match reviews with similar underlying sentiment (the data set comes with ground truth labels, labeling reviews as either positive or negative). As a part of the review is typically

[1] https://www.kaggle.com/datasets/atulanandjha/imdb-50k-movie-reviews-test-your-bert

Table 1: Benchmark statistics (number of rows in both input tables, average tuple sizes in tokens, and join selectivity).

Scenario	r_1	r_2	s_1	s_2	σ
Email	100	10	14	15	0.01
Reviews	50	50	98	101	0.5
Ads	16	16	11	10	0.06
Entailment	100	100	26	12	0.0035
Contradiction	100	100	26	12	0.0037
Words	10	1,000	3	3	0.05

sufficient to assess the underlying sentiment, longer reviews were shortened to the first 100 tokens. The join matches the first 50 reviews with the second 50 reviews, using the join condition "both reviews are positive or both are negative." The third scenario, "Ads," uses language models to match ads with corresponding searches, assuming that users enter their ads and requests via free text (e.g., on a platform like Craigslist). Ads are generated from the text template "Offering table that is [Material] and [Color]" and searches are generated from the template "Searching table that is [Material] and [Color]". Here, [Material] represents a specification of the material (e.g., "made of wood") and [Color] a specification of the color (e.g., "blue"). The next two scenarios are based on extracts from the MNLI data set, containing sentence pairs with entailment labels[2]. The join predicate either joins sentences where the first entail the second ("Entailment") or where the sentences contradict each other ("Contradiction"). The final scenario uses a sample of English words[3] for both input tables, matching pairs of words that start with the same letter.

4.2 Experimental Results

Table 1 reports statistics on the benchmarks, used for the experiments in this section. Note that overheads for semantic query processing are higher by many orders of magnitude, compared to the overheads of traditional, relational query processing. Hence, as shown in the following experiments, time and cost overheads are already non-negligible for some baselines, despite relatively small input sizes.

Figure 3 shows monetary execution fees, the number of input and output tokens, as well as execution time in seconds for all compared operator implementations and scenarios (as well as the average over all scenarios). Figure 4 shows precision, recall, and the F1 score when comparing output produced by different operator implementations to the ground truth.

The tuple nested loops join is the slowest join operator by far, averaging an execution time of over one hour over all scenarios (with up to two hours in some scenarios). Its average cost (19 cents) is higher by one order of magnitude, compared to all other joins except for the LOTUS implementation. This correlates with a high number of input read, as the tuple nested loops join reads each pair of input rows to evaluate the join condition. While this makes the tuple nested loops join impractical for larger input tables, it results in the highest F1 score over all baselines (58% on average). The join operator implementation used by LOTUS reduces run time significantly to 267 seconds while it incurs slightly higher average costs (38 cents) and a lower F1 score (30% on average). The embedding join improves time and cost over both aforementioned baselines, achieving minimal average costs and the lowest number of tokens read. On the other hand, it produces output of lower quality, obtaining the second-lowest F1 score of only 35% on average. Compared to other join algorithms, the embedding join

[2] https://www.kaggle.com/datasets/thedevastator/textual-entailment-dataset
[3] https://www.kaggle.com/datasets/jasperbutcher/words-csv

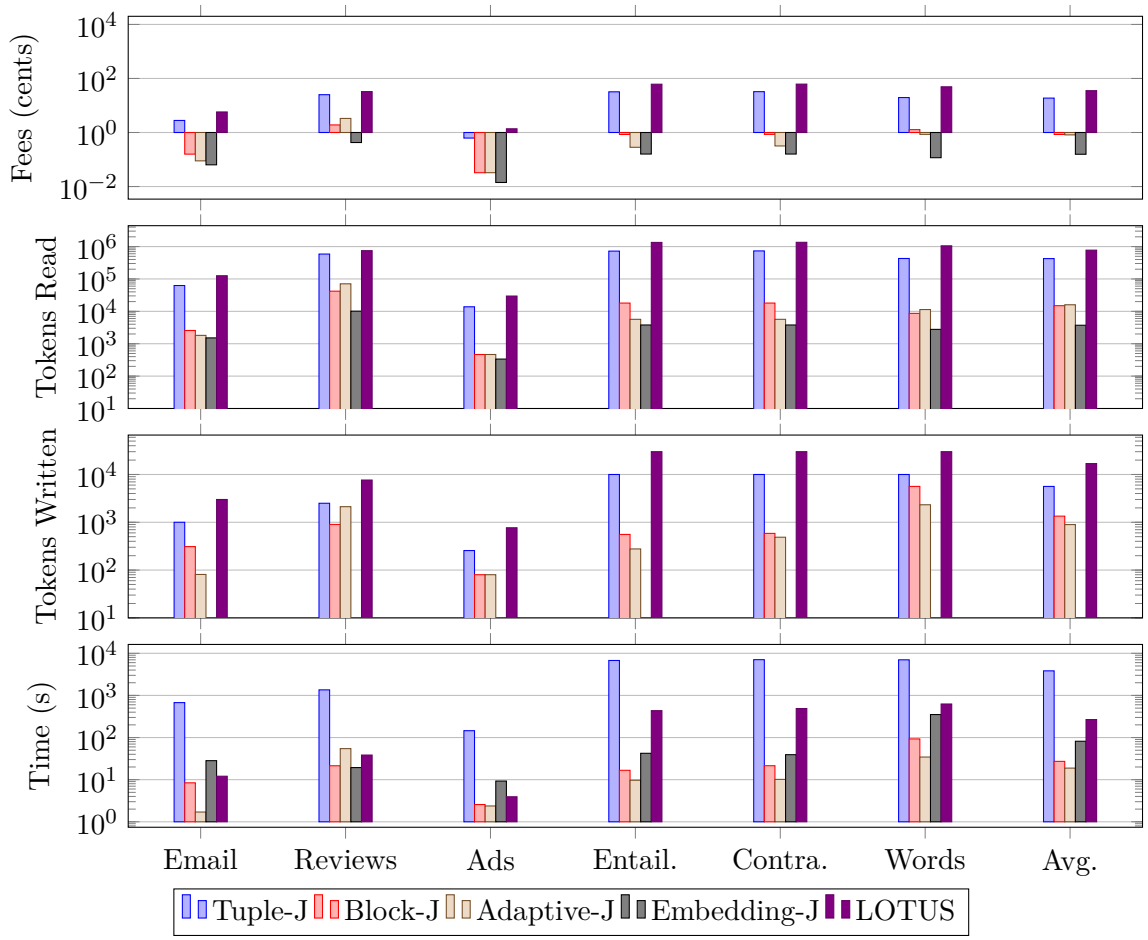

Figure 3: Cost of different join operators.

experiences a relatively high variance in result quality across different scenarios (achieving a perfect F1 score of 100% in some scenarios but a score of zero in others). For instance, the embedding join performs poorly for the Words scenario (F1 score of 3%), aimed at matching words with the same first letter. Here, similarity of embedding vectors, capturing semantic similarity, is a poor proxy for row pairs that satisfy the join condition. Similarly, the embedding join fails to find any matches in the Email scenario, aimed at finding contradicting statements. On the other hand, this join operator achieves a perfect F1 score of 100% in the Ads scenario, aimed at matching searches to suitable advertisements. Hence, while the embedding join can be useful in some scenarios, its result quality is entirely dependent on whether or not embedding vector similarity captures the semantics of the join predicate well.

The block nested loops join, as well as its adaptive variant, realize attractive tradeoffs between result quality and performance. They achieve minimal run time, with the adaptive algorithm realizing the minimal run time among all alternatives (19 seconds versus 28 seconds for the block nested loop variant). With the exception of the embedding join, they achieve minimal costs of less than one cent on average, improving over the most expensive algorithm by a factor of at most 50. The adaptive join algorithm improves over the block nested loops variant as it adapts the block size dynamically, enabling it to exploit scenarios in which few pairs of input rows satisfy the join condition. In those cases, few tokens need to be allocated for the output in the context window, enabling the algorithm to increase the size of the input blocks. Thereby, the number of LLM calls as well as the number of tokens read both decrease.

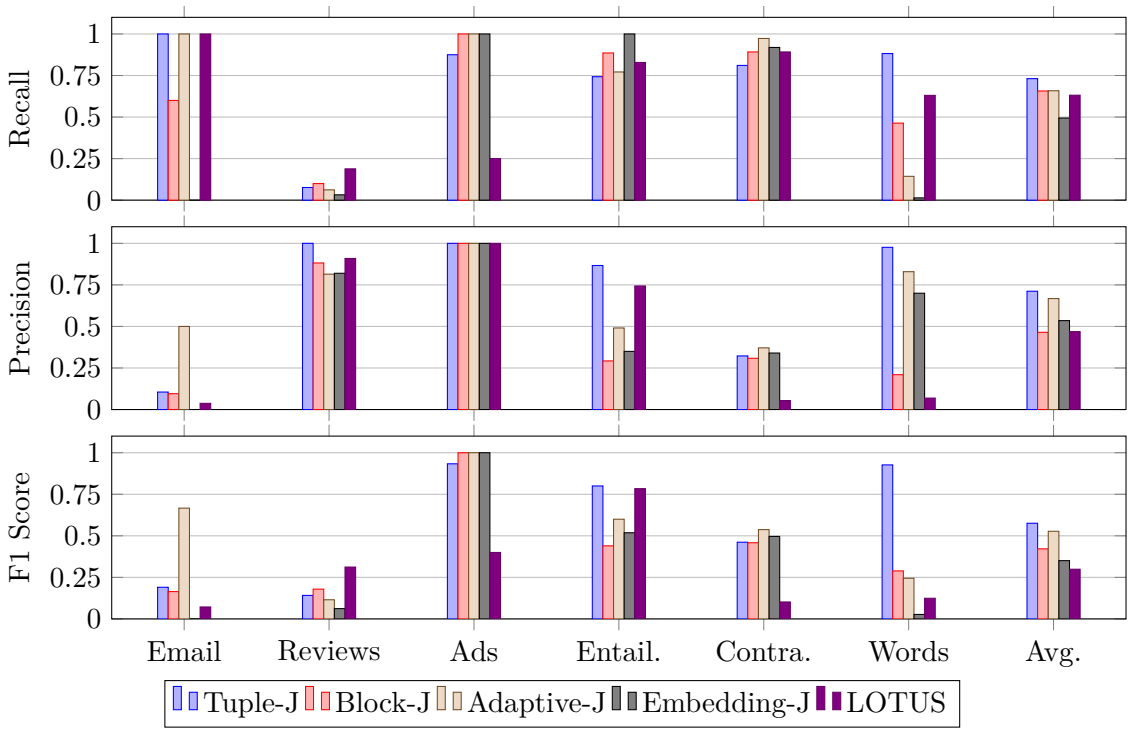

Figure 4: Output quality of different join operators.

At the same time, both join algorithms achieve near-optimal F1 scores. The adaptive join achieves the second-highest F1 score of 53%, close to the 58% realized by the tuple nested loops join. Interestingly, the adaptive variant achieves higher precision than the block variant in several scenarios. It seems that seeing more input rows at the same time (as the adaptive algorithm aims to maximize the size of the input batches), essentially a more representative sample, can sometimes help the LLM to identify matching rows more reliably.

5 Related Work

This work relates most to several recently proposed systems for semantic query processing [8, 9, 12, 13, 22, 29], enabling users to formulate queries that go beyond the capabilities of pure SQL. Many of those systems support variants of semantic join operators. For instance, Section 4 compares the proposed join operator implementations to the one used in the LOTUS system. The block-based join operator implementations described in this paper could be integrated into those systems as well. By its focus on implementing semantic versions of relational operators efficiently, this work relates to another recent paper [24]. In contrast to joins, the aforementioned paper focuses on efficient implementations of semantic sort operators.

This work connects to prior work that exploits language models for data management tasks [2, 4, 11, 18, 23, 26–28]. In particular, it connects to prior work leveraging language models for join processing [25]. However, prior work focuses on similarity-based joins (i.e., items match if they are more similar) and proposes a task-specific training phase. In contrast to that, the approach presented in this paper supports generic theta joins. The join condition is specified in natural language and may, in fact, connect tuples because they are dissimilar (e.g., matching tuples that represent contradicting statements, a scenario evaluated in Section 4). Also, unlike prior work requiring a task-specific training phase, the approaches

presented in this paper focus on a zero-shot scenario, avoiding the need for task-specific training labels. Different from other recent work [23], the approaches presented here assume that input data needs to be fed as input to the language model (rather than extracting information contained in the learned weights of the model).

As pointed out in a recent vision paper [21], implementing relational operators with language models connects to prior work leveraging crowdsourcing for data processing [6, 15, 19, 20]. In particular, it connects to prior work leveraging human crowd workers for joins and related matching tasks [5, 14, 16, 30, 31]. However, crowdsourcing adds specific challenges (e.g., the need to aggregate diverging answers from different crowd workers) whereas it removes others (e.g., hard bounds on the combined input and output size for each task), thereby motivating different algorithmic design decisions. Broadly, this work connects to prior approaches, adapting join algorithms to new processing contexts, e.g., multi-core architectures [1, 3], GPUs [10, 32], and FPGAs [7]. The approaches presented in this paper target a different platform (namely: language models) with unique properties.

6 Conclusion

This paper introduces, analyzes, and evaluates multiple variants of a novel implementation of the semantic join operator. Different from implementations used in current semantic query processing engines, this implementation integrates batches of rows into each prompt, thereby reducing the number of LLM invocations. This leads to significant performance advantages compared to prior operator implementations.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

References

[1] M. C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multicore database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[2] S. Arora, B. Yang, S. Eyuboglu, A. Narayan, A. Hojel, I. Trummer, and C. Re. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *PVLDB*, 17(2):92 – 105, 2023.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *Proceedings - International Conference on Data Engineering*, pages 362–373, 2013.

[4] Z. Chen, J. Fan, S. Madden, and N. Tang. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes. In *CIDR*, pages 1–7, 2023.

[5] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web*, pages 469–478, 2012.

[6] M. Franklin and D. Kossmann. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[7] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013*, pages 17–20, 2013.

[8] S. Jo and I. Trummer. Demonstration of ThalamusDB: Answering Complex SQL Queries with Natural Language Predicates on Multi-Modal Data. In *SIGMOD*, pages 179–182, 2023.

[9] S. Jo and I. Trummer. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *SIGMOD*, 2(3):1–26, 2024.

[10] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. *8th International Workshop on Data Management on New Hardware, DaMoN 2012 - In Conjunction with ACM SIGMOD/PODS Conference*, pages 55–62, 2012.

[11] M. Kayali, A. Lykov, I. Fountalis, N. Vasiloglou, D. Olteanu, and D. Suciu. CHORUS: Foundation Models for Unified Data Discovery and Exploration. *CoRR*, abs/2306.0, 2023.

[12] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, R. Shahout, and G. Vitagliano. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*, 2025.

[13] S. Madden, M. Cafarella, M. Franklin, and T. Kraska. Databases Unbound: Querying All of the World's Bytes with AI. *PVLDB*, 17(12):4564–4554, 2024.

[14] A. Marcus, E. Wu, and D. Karger. Human-powered sorts and joins. In *VLDB*, pages 13–24, 2011.

[15] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.

[16] A. Marcus, E. Wu, D. R. Karger, S. Madden, R. C. Miller, S. Acm, and N. York. Demonstration of Qurk : A query processor for human operators. In *SIGMOD*, pages 1315–1318, 2011.

[17] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[18] A. Narayan, I. Chami, L. Orr, and C. Ré. Can Foundation Models Wrangle Your Data? *PVLDB*, 16(4):738–746, 2022.

[19] A. G. Parameswaran. Human-Powered Data Management. page 225, 2013.

[20] A. G. Parameswaran, H. Park, H. Garcia-Molina, J. Widom, and N. Polyzotis. Deco: declarative crowdsourcing. In *Information and Knowledge Management*, pages 1203–1212, 2012.

[21] A. G. Parameswaran, S. Shankar, P. Asawa, N. Jain, and Y. Wang. Revisiting Prompt Engineering via Declarative Crowdsourcing. 2023.

[22] L. Patel, S. Jha, M. Pan, H. Gupta, P. Asawa, C. Guestrin, and M. Zaharia. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. In *Proceedings of the VLDB Endowment*, volume 18, pages 4171–4184, 2025.

[23] M. Saeed, N. De Cao, and P. Papotti. Querying Large Language Models with SQL. *CoRR*, abs/2304.0, 2023.

[24] F. Shao, J. Chen, Y. Pan, T. Rabbani, D. Agrawal, and A. E. Abbadi. Access Paths for Efficient Ordering with Large Language Models. *CoRR*, 2509.00303, 2025.

[25] S. Suri, I. Ilyas, C. Re, and T. Rekatsinas. Ember: No-Code Context Enrichment via similarity-based keyless joins. *PVLDB*, 15(3):699–712, 2021.

[26] J. Thorne, M. Yazdani, M. Saeidi, F. Silvestri, S. Riedel, and A. Halevy. From natural language processing to neural databases. *Proceedings of the VLDB Endowment*, 14(6):1033–1039, 2021.

[27] I. Trummer. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *PVLDB*, 15(11):2921 – 2928, 2022.

[28] I. Trummer. DB-BERT: a Database Tuning Tool that "Reads the Manual". In *SIGMOD*, pages 190–203, 2022.

[29] M. Urban and C. Binnig. CAESURA: Language Models as Multi-Modal Query Planners. In *CIDR*, 2024.

[30] W. Wang and M. Sebag. Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine Learning*, 92(2-3):403–429, 2013.

[31] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. In *Proceedings of the VLDB Endowment*, volume 6, pages 349–360, 2013.

[32] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.