

Implementing Semantic Join Operators Efficiently

Immanuel Trummer
Cornell University
Ithaca, USA
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). An adaptive variant of the proposed algorithm refers to cases in which the size of the output is difficult to estimate. 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.

ACM Reference Format:

Immanuel Trummer. 2018. Implementing Semantic Join Operators Efficiently. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Several recent systems [2, 6, 14, 21, 28, 32–34] 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06...\$15.00
<https://doi.org/XXXXXXX.XXXXXXX>

This paper focuses on a semantic version of a classical relational operator: the relational join. Semantic joins, as defined by systems like LOTUS [25], enable users to define the join condition in natural language. Note that this paper does not explicitly focus on equality joins. Instead, it focuses on general theta-joins [20] with natural language predicates. Such join operators are useful in the following example scenarios.

Example 1.1. To investigate a large corporation¹, 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.”

Example 1.2. A Website enables users to enter ads for used goods as free text, as well as free text descriptions of items they are searching for. Free text descriptions may contain detailed descriptions of various aspects of the items offered (or desired). For instance, a user looking for a new table may include constraints and preferences with regard to material, color, size, or state (e.g., “no coffee stains”). The Website wants to introduce a feature that supports users in matching ads to requests. This can be formulated as a join between two tables containing ads and searches. The join predicate can be expressed in natural language as “pairs of ads matching requests.”

Both tasks require natural language understanding as well as, potentially, some common-sense knowledge. E.g., determining that “I met Chris in Houston in the afternoon” contradicts “I saw Chris in Berlin at 1 PM” (assuming both statements refer to the same day and person) requires commonsense knowledge in terms of the minimal duration of a flight between the two locations. Fortunately, language models like OpenAI’s GPT models combine both capabilities and can be used, in principle, to evaluate the join predicates in the two aforementioned scenarios (as well as many others).

Perhaps the most natural way to use language models in the aforementioned scenarios is to compare pairs of entries from the two input collections. Iterating over all pairs of items from the two inputs, the language models can be tasked to compare two specific items with regard to the user-specified join condition. Essentially, this corresponds to a tuple nested loop join with the language model as predicate evaluation function (a corresponding approach is described in more detail in Section 3). To comply with database terminology, we will from now on refer to the two input collections as “tables” and to their elements as “tuples” (even if the items do not actually have to correspond to relational tuples).

¹This example is motivated by the investigation of the Enron corporation. In the context of this investigation, prosecutors had access to and analyzed some of over 500,000 emails from more than 150 employees.

The problem with the aforementioned approach is efficiency. Using LLMs is expensive. Providers such as OpenAI charge per the amount of text (measured in “tokens”, the atomic unit at which language models represent text) read and generated. Pairwise comparisons between a moderate number of 10,000 ads and searches with 100 tokens each (which corresponds to about a paragraph of text) using GPT-4 would cost about 600,000 dollars according to current rates². To make such approaches practical, it is crucial to reduce processing fees.

This paper shows that techniques from traditional database join algorithms [20, 26] can be adapted to improve efficiency for theta-joins, executed by language models, by many orders of magnitude. The core insight behind the algorithms presented in the following sections is the fact that language models can be used much more flexibly than pure predicate evaluation functions. More precisely, it is possible to task language models with finding matching pairs directly, within a set of tuples provided as part of the input. All it takes is a corresponding instruction in natural language, provided as part of the input as well. However, this does not mean that it is possible to use language models to perform the entire join operation in a single step. Language models come with strict constraints on the number of tokens that can be processed (i.e., read or generated) during a single invocation of the model. To scale the approach to larger data sets, it is necessary to decompose corresponding joins into multiple invocations of the LLM, each invocation only referring to a limited subset of the input data.

The approach described above resembles a block nested loops join. Similar to the classical join algorithm, the goal is to deal with limits on the data size that can be stored in the higher levels of the memory hierarchy, i.e., closer to the processor. In this scenario, the language model is the “processor” and size limits are imposed by the per-invocation token limit, intrinsic to the language model used. Despite those similarities, language models come with a very specific set of constraints, rendering a straightforward adaption of the traditional join operators inefficient.

For instance, for classical block nested loops joins, it is assumed that an output buffer of minimal size suffices. This assumption is justified if the output buffer can be repeatedly flushed to disk and re-filled during join processing, without losing the content of input buffers. However, in the case of language models, reaching the token limit when generating output tokens means that the request terminates. In that case, all input sent to the language model is lost and must be resent in a follow-up request (which incurs costs for reading input tokens again). This means, rather than generating join results gradually, we need to ensure that, for each invocation of the language model, the complete result fits within the token limit, determined by the token limit of the model, minus the number of tokens used up for the input data (and join task description).

The fact that a hard bound limits the combined size of the task description, (partial) join inputs, and join outputs is unique to the context of language models and requires careful planning. Sending too much input data within a single request is risky as the complete join result may exceed the token limit, possibly requiring redoing the corresponding task. On the other hand, sending less input than

possible is inefficient. It means that the language model is invoked more often than necessary. In the worst case, this approach reduces, essentially, to the tuple join, which uses language models to analyze single pairs of input tuples in each invocation.

The following sections develop, first of all, a custom cost model, calculating join processing costs as a function of input data properties (e.g., the number and average token size of the input tuples), language model properties (e.g., the cost per token read or generated and the maximal number of tokens processed per invocation), as well as of the batch sizes chosen for the two input tables (i.e., the number of tuples from each of the two input tables, sent per model invocation). The cost model focuses on fees paid for using the language model, typically the dominant cost factor when using LLMs such as GPT-4. Whereas data and model properties cannot be influenced, the number of tuples sent per model invocation can be chosen. Therefore, along with the cost model, the following sections derive formulas for calculating the optimal batch size for both input tables, given values for all relevant parameters describing data and model properties.

The cost model, and therefore the formulas for calculating optimal batch sizes, rely crucially on the selectivity of the join predicate. This selectivity determines how many join output tuples are generated, in expectation, per model invocation. Therefore, the selectivity determines how much “space,” i.e., how many tokens, need to be reserved for generating output as opposed to storing input tuples. The lower the selectivity, the fewer tokens need to be reserved for writing output. This means we can send more input tuples in each invocation of the language model, reducing the number of LLM invocations required to generate a complete join result (and therefore, as we will see in the following sections, the expected join costs).

As join predicates are formulated in natural language, it is not possible to apply standard methods to estimate their selectivity (e.g., based on histograms or other data statistics). However, it turns out that knowing the precise selectivity in advance is ultimately not necessary. This paper presents an adaptive join algorithm that automatically adapts join selectivity estimates, along with the associated choices for batch sizes. Starting from an optimistic selectivity estimate, i.e., an estimate that is possibly much smaller than the actual selectivity, the adaptive join algorithm starts by sending batches of tuples that may be too large to be processed in a single model invocation (since the amount of output generated exceeds the token limit). By a suitable design of the task instructions for the LLM, cases in which an incomplete result is generated due to the token limit can be recognized (we will use the term “overflow” in such cases). In the case of an overflow, the adaptive join algorithm updates the selectivity estimate by increasing it by a constant factor. Eventually, the selectivity reaches an estimate that is equal to or higher than the actual selectivity. This means that sending tuples does not result in an overflow anymore.

While it is clear that the adaptive join algorithm will eventually find a selectivity estimate that avoids overflows, it is not clear, a-priori, that this approach results in interesting performance properties. However, formal analysis shows that the adaptive join algorithm reaches near-optimal join processing costs under moderately simplifying assumptions.

²At the time of writing, OpenAI charges 3 cents per 1,000 tokens read when using GPT-4.

The experiments, using OpenAI’s GPT-4 model, demonstrate that batching tuples in join prompts leads to a dramatic reduction in semantic join costs. Specifically, the proposed join algorithms reduce processing overheads significantly compared to join algorithms used in LOTUS [25], a recently proposed semantic query processing engine. Comparing different join implementations proposed in this paper, it turns out that the block join works best if the selectivity of the join predicate is known. On the other hand, the adaptive version achieves nearly the same performance without requiring a selectivity value beforehand. A simple approach exploiting embedding vectors to match row pairs during the join works best in scenarios where the join condition is semantically close to an equality join. In scenarios where the goal is to match items that are complementary (e.g., matching contradicting statements), the result quality may, however, degrade severely.

In summary, the original scientific contributions in this paper are the following:

- The paper introduces multiple algorithms implementing semantic joins with arbitrary (i.e., not necessarily equality) join conditions, described in natural language.
- The paper analyzes the cost of the proposed algorithms in terms of token consumption, proposing formulas to tune these implementations for optimal performance.
- The paper presents experiments, evaluating the proposed algorithms in several scenarios, comparing to multiple baseline algorithms (some of which are currently used in semantic query processing engines).

The remainder of this paper is organized as follows. Section 2 introduces the problem model and related terminology. Section 3 describes a simple join algorithm that uses language models for pairwise tuple comparisons. Section 4 describes a join operator that exploits LLMs for finding matching pairs between tuple batches. Section 5 shows how to optimize batch sizes for that join operator if the selectivity of the join predicate is known. Section 6 presents an adaptive join operator that automatically updates selectivity estimates while achieving near-optimal performance. Section 7 reports on experiments, comparing all join operators in different scenarios and according to different metrics. Finally, Section 8 contrasts the work presented in this paper with prior work.

2 PROBLEM MODEL

This paper addresses the following problem.

Definition 2.1 (Semantic Join with Natural Language Predicates). Given two tables R_1 and R_2 , together with a join predicate j , expressed as free text in natural language, find all pairs $R \subseteq R_1 \times R_2$ that satisfy predicate j .

Tuples may represent text documents or a textual representation of structured records. The aforementioned problem can be solved by LLMs.

Definition 2.2 (Large Language Model). A large language model processes arbitrary tasks, described in natural language in the prompt (the input text sent to the model). Processing fees are proportional to the number of tokens (the atomic unit at which text is represented) read and generated (with possibly different cost factors

Table 1: Symbols and their semantics.

Symbol	Semantics
r_i	Number of rows in table i
b_i	Number of rows per batch for table i
$b_i^*(\sigma)$	Optimal batch size for table i
s_i	Token size per entry in table i
σ	Selectivity of join condition
g	Relative cost of generating tokens
p	Size of task description with predicate
t	Token threshold per LLM invocation
$c(b_1, b_2)$	Total processing costs
$c^*(b_1)$	Cost for given b_1 and optimal choice of b_2
$o(e, \sigma)$	Join cost with selectivity σ when optimizing for e

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

```

1: // Perform tuple join between relations  $R_1$  and  $R_2$ ,
2: // using join condition  $j$ .
3: function BLOCKJOIN( $R_1, R_2, j$ )
4:   // Initialize result set
5:    $R \leftarrow \emptyset$ 
6:   // Iterate over tuple pairs
7:   for  $t_1 \in R_1$  do
8:     for  $t_2 \in R_2$  do
9:       // Create prompt for LLM
10:       $P \leftarrow \text{TUPLEPROMPT}(t_1, t_2, j)$ 
11:      // Ask LLM if join condition satisfied
12:       $A \leftarrow \text{INVOKELLM}(P)$ 
13:      // Add result tuple if answer is positive
14:      if  $A = \text{"Yes"}$  then
15:         $R \leftarrow R \cup \{ \langle t_1, t_2 \rangle \}$ 
16:      end if
17:    end for
18:  end for
19:  // Return join result
20:  return  $R$ 
21: end function

```

for tokens read and generated). The sum of tokens read and generated per model invocation is upper-bounded by a model-specific constant.

Table 1 summarizes all symbols introduced in the next sections.

3 TUPLE NESTED LOOPS JOIN

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

3.1 Algorithm

Algorithm 1 shows the tuple nested loops join algorithm, adapted to use a large language model to evaluate join conditions. The input to Algorithm 1 are the two tables, R_1 and R_2 , as well as the join condition, j . The join condition is formulated in natural language and expresses the condition for a match between two tuples. As

```

Is the following true ("Yes"/"No"): [j]?
Text 1: [t1]
Text 2: [t2]
Answer:

```

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

the classical tuple nested loops join, Algorithm 1 iterates over all combinations of tuples from the input tables. The particularity of Algorithm 1 lies in the way the join condition is evaluated.

To evaluate a join condition, Algorithm 1 performs three steps. First, it generates a prompt, instructing the language model to compare the two current tuples. Second, it invokes a language model with that prompt to execute that comparison. Finally, it interprets the text answer by the language model, adding the tuple combination to the result set if the two input tuples match.

Figure 1 shows the template used for generating prompts. It contains several placeholders, marked by square brackets. Function `TUPLEPROMPT`, used in Algorithm 1, instantiates this template by substituting placeholders with values from the input parameters. The start of the prompt template describes the task to the language model (answering the question of whether or not the following condition holds), as well as the desired output format (i.e., either a “Yes” or a “No”). The instructions contain a placeholder for the join condition, `[j]`, describing the conditions for a match. After that, the prompt contains the data, i.e., the two tuples to compare (represented via placeholders `[t1]` and `[t2]`). The prompt concludes with a request for an answer, indicating to the language model that all relevant information for the task has been conveyed.

Function `INVOKELLM` submits prompts to a language model (e.g., GPT-4) and returns the generated answer. In principle, the generated answer could be arbitrary text. However, as the prompt specifies an expected output format, the answer should be either “Yes” or “No” in most cases. Any valid answer uses one single token. For that reason, the implementation of `INVOKELLM` configures the language model to generate at most one single output token (thereby avoiding rare but costly cases in which the language model might generate a longer text as a reply, misunderstanding the instructions).

3.2 Cost Model

The following cost model estimates (monetary) processing costs as a function of input properties. Parameters r_1 and r_2 denote the number of rows in the two input tables. Parameters s_1 and s_2 denote the (average) sizes of a tuple in the two input tables, measured in terms of the number of tokens (since Cloud providers of language models such as OpenAI charge per token processed). Also, p denotes the number of tokens used for the part of the prompt that remains static across different loop iterations (i.e., all text except for the compared tuples). In some cases, generating output is more expensive than generating input. Parameter g denotes the relative cost overhead of generating tokens, compared to reading tokens.

LEMMA 3.1. *Comparing two input tuples incurs cost $p + s_1 + s_2 + g$.*

Algorithm 2 Block nested loops join algorithm for semantic joins, executed via large language models.

```

1: // Perform block join between relations  $R_1$  and  $R_2$ 
2: // with join condition  $j$  and using block sizes  $b_1$  and  $b_2$ .
3: function BLOCKJOIN( $R_1, R_2, j, b_1, b_2$ )
4:   // Initialize result set
5:    $R \leftarrow \emptyset$ 
6:   // Partition input into batches
7:    $\mathcal{B}_1 \leftarrow \{B_i \subseteq R_1 \mid R_1 = \dot{\cup}_i B_i, \forall i |B_i| = b_1\}$ 
8:    $\mathcal{B}_2 \leftarrow \{B_i \subseteq R_2 \mid R_2 = \dot{\cup}_i B_i, \forall i |B_i| = b_2\}$ 
9:   // Iterate over pairs of batches
10:  for  $B_1 \in \mathcal{B}_1$  do
11:    for  $B_2 \in \mathcal{B}_2$  do
12:      // Create prompt for LLM
13:       $P \leftarrow \text{BLOCKPROMPT}(B_1, B_2, j)$ 
14:      // Get raw answer from LLM
15:       $A \leftarrow \text{INVOKELLM}(P)$ 
16:      // Check for overflow
17:      if  $A[-1] \neq \text{Finished}$  then
18:        return <Overflow>
19:      end if
20:      // Extract result tuples
21:       $R \leftarrow R \cup \text{EXTRACTTUPLES}(B_1, B_2, A)$ 
22:    end for
23:  end for
24:  // Return join result
25:  return  $R$ 
26: end function

```

PROOF. Tuple-independent parts of the prompt account for p tokens read. In addition, the information about the two input tuples, i.e., $s_1 + s_2$ tokens must be read. Finally, one output token (“Yes” or “No”) is generated in each iteration with cost g . $\square$

Total join processing costs follow immediately.

COROLLARY 3.2. *Join processing costs are $r_1 \cdot r_2 \cdot (p + s_1 + s_2 + g)$.*

PROOF. This follows from the cost per comparison (Lemma 3.1) and the number of comparisons, determined by the number $r_1 \cdot r_2$ of iterations of the innermost nested loop. $\square$

4 BLOCK NESTED LOOPS JOIN

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

4.1 Algorithm

Algorithm 2 uses similar input parameters as Algorithm 1, namely two input tables (R_1 and R_2) and a join condition j . In addition, Algorithm 2 uses input parameters b_1 and b_2 , representing the number of tuples from the first and second table that are processed together as one batch. The choice of those parameter values is non-trivial and analyzed in the following sections.

Algorithm 2 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

```

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 2: Prompt template used for block nested loops join (instantiated by Function BLOCKPROMPT in pseudo-code).

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 2 invokes the model only once for each pair of tuple batches.

Figure 2 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 $[B_i[j]]$ 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 2 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 2 sends prompts generated for the current pair of batches to the language model to retrieve an answer. First of all, Algorithm 2 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. Section 6 shows how to handle such cases. If no overflow occurs, the tuples associated with the index pairs are added to the result set. Function EXTRACTTUPLES (the pseudo-code is omitted due to space restrictions) translates index pairs in the answer into tuple pairs.

4.2 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 2 (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 2). 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 4.1. *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

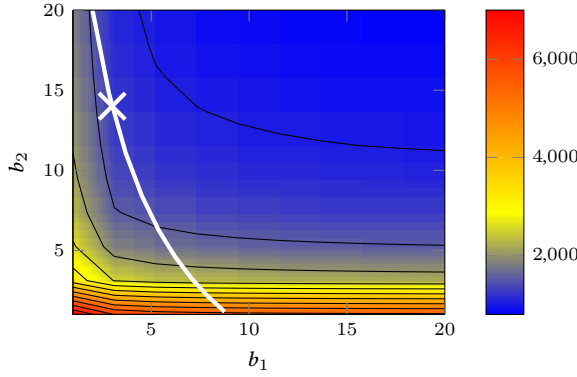


Figure 3: Illustrating join processing costs as a function of the two input batch sizes (b_1 and b_2), using $r_1 = 50$, $r_2 = 10$, $s_1 = 10$, $s_2 = 2$, $s_3 = 1$, $\sigma = 1$, $g = 1$, $p = 1$. All solutions under the white curve use prompts with a size at or below 100 tokens. The white X marks the solution with minimal cost among all solutions with a prompt size of up to 100 tokens.

taking into account tokens required for the join task description (p) yields the postulated size formula. $\square$

LEMMA 4.2. *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 4.1. Costs are proportional to the number of tokens, except that it distinguishes tokens read from generated tokens. The LLM only generates tokens associated 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. $\square$

LEMMA 4.3. *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 2. 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. $\square$

COROLLARY 4.4. *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 4.2 and 4.3, obtained by multiplying the cost per LLM invocation with the number of LLM invocations. $\square$

5 OPTIMIZING FOR KNOWN SELECTIVITY

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 5.1. Figure 3 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 3 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 3, those are the points below the white line) while minimizing costs under that constraint. The white X marks the optimal solution in Figure 3.

5.1 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.

$$b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma \leq t \quad (1)$$

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 5.2. *Maximizing the number of tokens processed per LLM invocation minimizes processing costs.*

PROOF. Assume that the prompt size is below the threshold, i.e., $b_1 \cdot s_1 + b_2 \cdot s_2 + b_1 \cdot b_2 \cdot s_3 \cdot \sigma < t$. Furthermore, 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. $\square$

Example 5.3. Consider the cost function depicted in Figure 3. As discussed before, the white curve marks points at which the number of tokens processed per LLM invocation equals the threshold. Due to Theorem 5.2, 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 5.4. *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 5.2, 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 $b_2 = (t - b_1 \cdot s_1)/(s_2 + b_1 \cdot s_3 \cdot \sigma)$ $\square$

According to Lemma 5.4, substituting each occurrence of b_2 in the join cost function with $b_2(b_1)$ yields minimal processing costs:

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

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 ($c^*(b_1)$).

5.2 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:

$$\frac{dc^*}{db_1}(b_1^*) = 0 \quad \frac{d^2c^*}{db_1^2}(b_1^*) > 0$$

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

$$\frac{dc^*}{db_1} = r_1 r_2 (t + p) \left[\frac{b_1^2 s_1 s_3 \sigma + b_1 2s_1 s_2 - s_2 t}{(t - b_1 s_1)^2 b_1^2} \right] \quad (2)$$

LEMMA 5.5. 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 $b_1^2 s_1 s_3 \sigma + b_1 2s_1 s_2 - s_2 t = 0$. This is a quadratic equation in b_1 . The roots are therefore given by $(-2s_1 s_2 \pm \sqrt{(2s_1 s_2)^2 - 4(s_1 s_3 \sigma)(-s_2 t)}) / (2s_1 s_3 \sigma)$ 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 $s_1 s_2 < \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}$. $\square$

THEOREM 5.6. 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 $d^2c^*/db_1^2 > 0$ at b_* since b_* is a critical point, according to Lemma 5.5. Set $u(b_1) = b_1^2 s_1 s_3 \sigma + b_1 2s_1 s_2 - s_2 t$ 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 2. Due to the quotient rule, it is $d^2c^*/db_1^2 = r_1 r_2 (t + p) [u'v - uv'] / v^2$ where $u' = du/db_1$ and $v' = dv/db_1$. As outlined in the proof of Lemma 5.5, $u(b_*) = 0$. Hence, at b_* , the second-order derivative d^2c^*/db_1^2 simplifies to $r_1 r_2 (t + p) [u'v] / v^2$. It is $u' = d/db_1 [b_1^2 s_1 s_3 \sigma + b_1 2s_1 s_2 - s_2 t] = 2b_1 s_1 s_3 \sigma + 2s_1 s_2$. As all constants appearing in this equation are positive with $s_1 > 0$ and $s_2 > 0$, u' is strictly positive

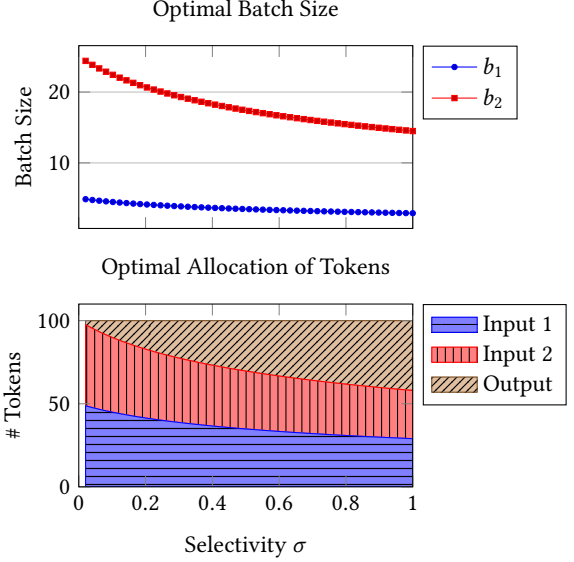


Figure 4: Impact of selectivity σ on optimal batch sizes and token allocations for $r_1 = 50$, $r_2 = 10$, $s_1 = 10$, $s_2 = 2$, $s_3 = 1$, $g = 1$, $p = 1$, and $t = 100$.

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^2c^*/db_1^2 is greater than zero at b_* . $\square$

Example 5.7. In the example depicted in Figure 3, 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 5.4, 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 3, the white X marks that point.

6 ADAPTIVE JOIN ALGORITHM

The previous section optimizes batch sizes, assuming that the selectivity σ of the join predicate is known. This section relaxes that assumption and shows how to deal with an unknown selectivity.

6.1 Algorithm

The following example illustrates the impact of selectivity.

Example 6.1. Figure 4 demonstrates the impact of the join predicate selectivity. With the exception of the selectivity estimate, σ , the

Algorithm 3 Adaptive join algorithm, updating selectivity estimates as needed.

```

1: // Perform block join with between relations  $R_1$  and  $R_2$ 
2: // using join condition  $j$  with optimistic selectivity estimate  $e$ .
3: function ADAPTIVEJOIN( $R_1, R_2, j, e$ )
4:   // Generate data size statistics
5:    $stats \leftarrow \text{GENERATESTATISTICS}(R_1, R_2, j)$ 
6:   // Initialize join result to overflow flag
7:    $R \leftarrow \text{<Overflow>}$ 
8:   // Iterate until complete join result available
9:   while  $R = \text{<Overflow>}$  do
10:    // Calculate optimal batch sizes
11:     $\langle b_1, b_2 \rangle \leftarrow \text{OPTIMALBATCHSIZES}(stats, e)$ 
12:    // Try block join with those sizes
13:     $R \leftarrow \text{BLOCKJOIN}(R_1, R_2, j, b_1, b_2)$ 
14:    // Increase selectivity estimate
15:     $e \leftarrow e \cdot \alpha$ 
16:  end while
17:  // Return join result
18:  return  $R$ 
19: end function

```

example uses the same settings as in Figure 3. The upper plot shows the optimal settings for the batch sizes, b_1 and b_2 , as a function of selectivity (on the x-axis). The lower plot shows how tokens read or written in each LLM invocation are partitioned across tokens representing input from the first and the second table, as well as output tuples (which are written by the LLM). In the example, a higher selectivity motivates smaller batches (the analysis in the following subsection shows that this, as well as other observations from the example, generalize). Intuitively, this makes sense as the number of output tuples increases in the selectivity. Hence, keeping batch sizes constant while selectivity increases leads to an overflow, i.e., the size required for join output exceeds the maximal number of tokens. Consistent with that, the number of tokens reserved for join output increases, relative to tokens reserved for representing input, as selectivity increases.

The example shows that optimal choices for batch sizes depend significantly on selectivity. Traditional selectivity estimation methods, based on data statistics, **cannot be used** for join predicates in natural language. At the same time, using a selectivity estimates that deviates significantly from the actual selectivity has negative consequences.

Using a selectivity estimate that is too high is inefficient. More precisely, assuming selectivity that is too high means reserving more tokens for join output than necessary. Those tokens could be used for representing more input tuples, thereby reducing the number of iterations and, ultimately, costs. On the other hand, using a selectivity estimate that is too low is ineffective. If not reserving enough tokens for join output, the language model will be unable to generate a complete join result. In that case, the block join algorithm (Algorithm 2) returns the **<Overflow>** flag, indicating an incomplete join result.

Fortunately, it turns out that an adaptive processing strategy, shown in Algorithm 3, achieves near-optimal performance, despite

not assuming a precise selectivity estimate. Algorithm 3 starts with an optimistic selectivity estimate, i.e., a selectivity estimate that is assumed to be lower than the actual selectivity. Choosing an estimate that is closer to the actual selectivity may improve performance but the effect is bounded, as shown by the analysis in the following subsection. It is, in principle, possible to start with a **higher** (i.e., pessimistic) selectivity estimate and **decrease** it to match the actual selectivity more closely. This approach is equivalent if selectivity is constant across different batches, ensuring that the selectivity observed on a sample is representative. However, in practice, selectivity differs across batches due to data skew. Hence, after lowering the selectivity estimate, meaning that less space is reserved for output in the prompt, it may be necessary to increase estimates again to avoid overflow if later batches have a higher selectivity. However, selectivity updates are undesirable as they cause overheads. When only increasing selectivity estimates, meaning that more and more space in the prompt is reserved to store output, it is never necessary to revert prior decisions and decrease estimates again to ensure that the join operator can finish (i.e., all batches are processed without overflow).

Algorithm 3 calculates all relevant data statistics that appear in the formulas from Section 5. For instance, this includes the average token sizes of input tuples from both tables, as well as their cardinality. After that, Algorithm 3 iterates until a complete join result is generated. As a sub-function, it uses the block join algorithm, presented in Section 2. Batch sizes are calculated, based on the current selectivity estimate. Function OPTIMALBATCHSIZES encapsulates the formulas for calculating optimal batch sizes, derived in Section 5. If the block join algorithm returns the **<Overflow>** flag, the selectivity estimate is increased by a factor of α . Factor $\alpha > 1$ is a tuning parameter, its impact is studied in the next subsection.

6.2 Analysis

The following lemmata establish properties of the optimal batch size for the first table as a function of selectivity: $b_1^*(\sigma)$.

LEMMA 6.2. *The optimal value for the batch size in the first table with selectivity σ , $b_1^*(\sigma)$, is anti-monotone in the selectivity σ .*

PROOF. According to Theorem 5.6, it is

$$b_1^*(\sigma) = \frac{\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} - s_1 s_2}{s_1 s_3 \sigma}.$$

Multiplying numerator and denominator by $(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} + s_1 s_2)$ yields

$$\begin{aligned}
b_1^*(\sigma) &= \frac{(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} - s_1 s_2)(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} + s_1 s_2)}{(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} + s_1 s_2)s_1 s_3 \sigma} \\
&= \frac{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t - s_1^2 s_2^2}{(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} + s_1 s_2)s_1 s_3 \sigma} \\
&= \frac{s_2 t}{(\sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t} + s_1 s_2)}.
\end{aligned}$$

As the numerator does not depend on σ , while the denominator is monotone in σ , this fraction and therefore $b_1^*(\sigma)$ is anti-monotone in the selectivity σ . $\square$

LEMMA 6.3. *If $e \geq \sigma \geq e/\alpha$ then $b_1^*(\sigma) \leq \alpha \cdot b_1^*(e)$.*

PROOF. The following holds due to Theorem 5.6 and $e/\alpha \leq \sigma$:

$$\begin{aligned} \alpha \cdot b_1^*(e) &= \alpha \cdot [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 e t}] / (s_1 s_3 e) \\ &= [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 e t}] / (s_1 s_3 (e/\alpha)) \\ &\geq [-s_1 s_2 + \sqrt{s_1^2 s_2^2 + s_1 s_2 s_3 \sigma t}] / (s_1 s_3 \sigma) \\ &= b_1^*(\sigma) \end{aligned}$$

$\square$

The following lemma analyzes the product of optimal batch sizes as a function of selectivity, denoted as $b_1^*(\sigma)$ and $b_2^*(\sigma)$.

LEMMA 6.4. *If $e \geq \sigma \geq e/\alpha$ then $b_1^*(\sigma) \cdot b_2^*(\sigma) \leq \alpha \cdot b_1^*(e) \cdot b_2^*(e)$.*

PROOF. The proof uses contradiction. Assume that $b_1^*(\sigma) \cdot b_2^*(\sigma) > \alpha \cdot b_1^*(e) \cdot b_2^*(e)$. According to Lemma 6.3, it is $b_1^*(\sigma) \leq \alpha \cdot b_1^*(e)$. Therefore, $b_1^*(\sigma) \cdot b_2^*(\sigma) > \alpha \cdot b_1^*(e) \cdot b_2^*(e)$ implies $b_2^*(\sigma) > b_2^*(e)$. Due to Theorem 5.2, assuming selectivity e , the optimal values for b_1 and b_2 exploit the full number of tokens:

$$b_1^*(e) \cdot s_1 + b_2^*(e) \cdot s_2 + b_1^*(e) \cdot b_2^*(e) \cdot s_3 \cdot e = t$$

However, exploiting $b_2^*(\sigma) > b_2^*(e)$ and $b_1^*(\sigma) \cdot b_2^*(\sigma) > \alpha \cdot b_1^*(e) \cdot b_2^*(e)$, then anti-monotonicity of $b_1^*(\sigma)$, according to Lemma 6.2, with $e \geq \sigma \geq e/\alpha$, and, finally, $\alpha > 1$, yields:

$$\begin{aligned} &b_1^*(\sigma) \cdot s_1 + b_2^*(\sigma) \cdot s_2 + b_1^*(\sigma) \cdot b_2^*(\sigma) \cdot s_3 \cdot \sigma \\ &> b_1^*(\sigma) \cdot s_1 + b_2^*(e) \cdot s_2 + \alpha \cdot b_1^*(e) \cdot b_2^*(e) \cdot s_3 \cdot \sigma \\ &\geq b_1^*(e) \cdot s_1 + b_2^*(e) \cdot s_2 + b_1^*(e) \cdot b_2^*(e) \cdot s_3 \cdot e = t \end{aligned}$$

This leads to a contradiction since the number of tokens used with selectivity σ exceeds the number t of available tokens. $\square$

Denote by $o(e, \sigma)$ the join processing costs when optimizing for selectivity estimate e while the actual selectivity is σ .

THEOREM 6.5. *If $e \geq \sigma \geq e/\alpha$ then $o(e, \sigma) \leq \alpha \cdot g \cdot o(\sigma, \sigma)$.*

PROOF. Optimizing for an estimated selectivity of e , the number of model invocations for optimal batch sizes is $r_1 \cdot r_2 / (b_1^*(e) \cdot b_2^*(e))$, according to Lemma 4.3. According to Lemma 6.4, it is $b_1^*(\sigma) \cdot b_2^*(\sigma) \leq \alpha \cdot b_1^*(e) \cdot b_2^*(e)$. Therefore, the number of model invocations when optimizing for selectivity e , rather than actual selectivity σ , is higher at most by factor α : $r_1 \cdot r_2 / (b_1^*(e) \cdot b_2^*(e)) \leq \alpha \cdot r_1 \cdot r_2 / (b_1^*(\sigma) \cdot b_2^*(\sigma))$. Processing costs are proportional to the number of model invocations and the cost per invocation. According to Theorem 5.2, any optimal choice for batch sizes leads to prompts that exploit the maximal number of tokens. The cost per prompt is therefore between t (if all tokens are read) and $t \cdot g$ with $g \geq 1$ (if all tokens are written). Hence, optimizing for estimated selectivity e , rather than selectivity σ , can increase per-invocation costs at most by factor g . The postulated bound follows since the number of model invocations increases at most by factor α and the cost per invocation at most by factor g . $\square$

The following theorem bounds join processing costs, assuming imprecise selectivity estimates.

THEOREM 6.6. *Given constant tuple sizes and ratios between actual and initial estimated selectivity, Algorithm 3 converges to cost within factor $\alpha \cdot g$ of the optimum as the size of the input data grows.*

PROOF. Assuming constant tuple sizes in both input tables, using batch sizes that are too large immediately results in an overflow (i.e., Algorithm 2 returns **<Overflow>** after a single invocation of the LLM). This means after $O(\log_{\alpha}(\sigma/e))$ LLM invocations, the selectivity estimate e has been adapted to be at least as large as the actual selectivity σ . As e and σ are assumed constant and the maximal overhead per LLM invocation is bounded by constants as well ($t \cdot g$), the overheads due to incorrect selectivity estimates are constant as well. As the data size grows, the overheads of join processing with an estimate $e \geq \sigma$ become dominant. Also, since Algorithm 3 updates estimates via multiplication by factor α , it is $\sigma \geq e/\alpha$. According to Theorem 6.5, the cost overhead is therefore upper-bounded by factor $\alpha \cdot g$. $\square$

7 EXPERIMENTAL RESULTS

The following experiments evaluate the join operators. Section 7.1 describes the experimental setup. Section 7.2 reports on the results of simulated joins, showing how costs of different operator implementations scale as a function of the input size. Section 7.3 reports on the results of an evaluation that uses OpenAI's GPT-4 model and compares the approaches proposed in this paper to multiple baselines.

7.1 Experimental Setup

The following experiments use a simulator as well as experiments with real LLMs. The simulator is implemented in Python 3.11. It goes beyond applying the formulas, presented in the previous sections, and simulates each single prompt instead. Unless noted otherwise, the simulation assumes a maximal context size of 8,192 tokens, a join predicate selectivity of $\sigma = 0.001$, input tuple sizes of 30 tokens (i.e., $s_1 = s_2 = 30$, this corresponds to a few sentences of text), two tokens per output tuple (i.e., $s_3 = 2$), and a tuple-independent prompt size of 50 tokens (i.e., $p = 50$). To translate token counts into processing fees, it uses the pricing of the GPT-4 default model by OpenAI. At the time of writing, the default version charges 3 cents per 1,000 tokens read and 6 cents per 1,000 tokens generated (i.e., the relative cost of writing tokens, g , is two). By default, each table contains $r_1 = r_2 = 5,000$ tuples (some experiments use larger tables, this is pointed out in the text). It is $\alpha = 4$ for the adaptive join.

Beyond simulation, the experiments use OpenAI's GPT-4 model (gpt-4-0613). Join operators are implemented in Python 3.11, using OpenAI's Python client in version 1.12. GPT-4 is invoked with a per-request timeout of 20 seconds. The temperature parameter of GPT-4 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"). Unless noted otherwise, GPT-4 is used with a maximal context size of 2,000 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 [25], 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 three scenarios, connected to the use cases discussed in the introduction. The project code repository³ contains data generation scripts for all of the following benchmarks. The “Emails” 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 Kaggle⁴. 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 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”).

7.2 Simulation Results

Figure 5 compares processing costs of different join operator implementations, varying the size of the first input table, the size of the tuples (s_1), as well as the selectivity of the join predicate (σ). It compares the tuple join (Algorithm 1), the block join (Algorithm 2) when calculating batch sizes for a conservative selectivity estimate of one (which ensures enough space for result output), abbreviated as “Block-C”, and the same algorithm when calculating batch sizes informed by the actual selectivity, abbreviated as “Block-I”. Finally, it reports results for the adaptive join algorithm (Algorithm 3), using an optimistic selectivity estimate of $\sigma/100$ for each benchmark (i.e., initially underestimating selectivity by factor 100). The y-axis of Figure 5 is logarithmic.

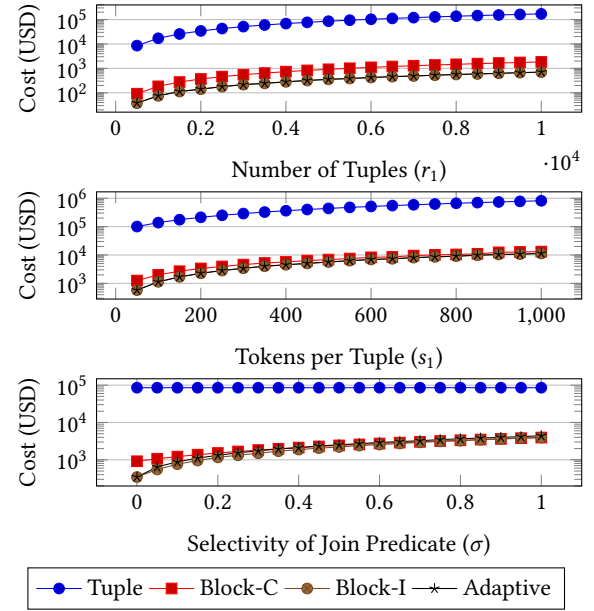


Figure 5: Cost of simulated joins with GPT-4.

The costs of the tuple join are higher than the costs of the other join operators by several orders of magnitude. E.g., joining tables containing 10,000 and 5,000 tuples costs over 100,000 dollars when using the tuple join but less than 1,000 dollars for the Adaptive join. Among the other join operators, the block join with conservative selectivity estimates (Block-C) performs worse than the one with accurate selectivity estimates (Block-I). For instance, for an input size of 10,000 tuples, Block-C is about three times more expensive than Block-I. Block-I is difficult to implement as it requires precise selectivity estimates (which would require additional profiling mechanisms that incur additional costs). However, the adaptive algorithm performs almost identical to Block-I (e.g., cost within 0.1% of Block-I for 10,000 input tuples) and does not require accurate selectivity estimates, making it the most practical alternative.

Increasing the number of input rows, tuple size, or join selectivity increases processing overheads for almost all operators. An exception is the tuple join for which costs do not increase when increasing join selectivity. This is expected as, unlike for the block join variants, the tuple join generates the same amount of output for matching tuple pairs as for non-matching tuple pairs. The gap between different block join variants (i.e., Block-C, Block-I, and also Adaptive) varies as a function of scenario properties. As selectivity increases, the (pessimistic) assumptions on the selectivity, underlying tuning choices made by Block-C, become accurate. Hence, the gap between block join variants shrinks as selectivity increases.

7.3 Benchmarks with Real LLMs

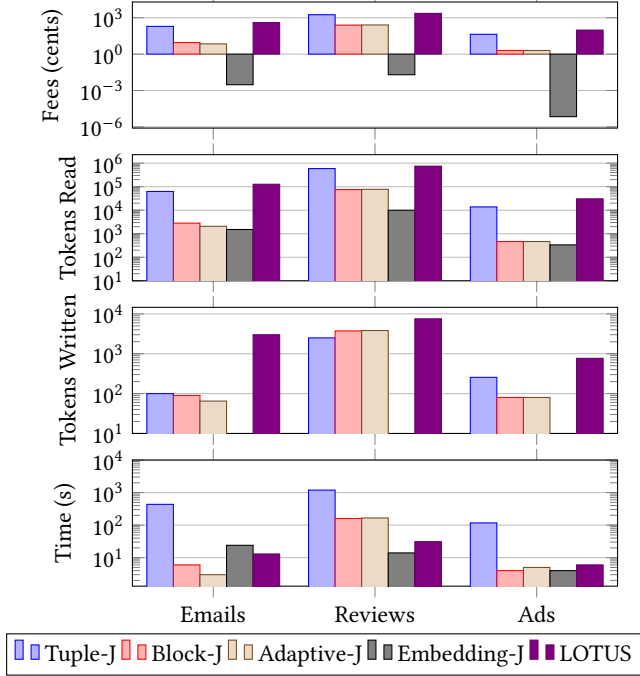
Table 2 reports statistics on the benchmarks, used for the experiments in this section. Figure 6 reports the cost of different join operators incurred in (non-simulated) experiments with GPT-4. As in the simulation, the execution costs for the tuple join are higher than the costs for the block join variants by orders of magnitude.

³<https://github.com/itrummer/llmjoins>

⁴<https://www.kaggle.com/datasets/atulanandjha/imdb-50k-movie-reviews-test-your-bert>

Table 2: Benchmark statistics.

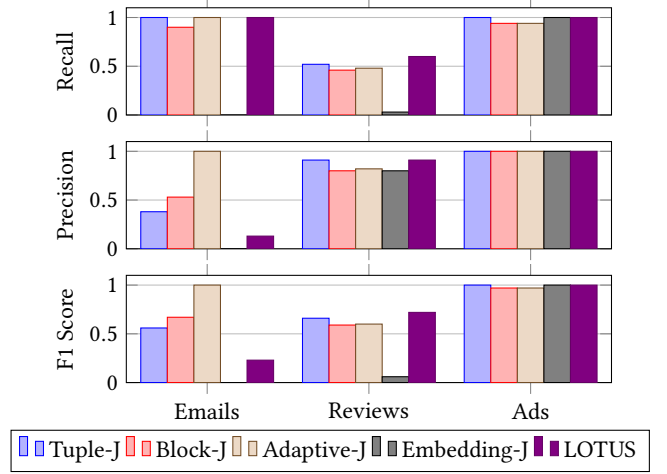
Property	Emails	Reviews	Ads
Tbl 1 Rows	100	50	16
Tbl 2 Rows	10	50	16
Tbl 1 Avg. Tuple Size	14	98	11
Tbl 2 Avg. Tuple Size	15	101	10
Predicate Selectivity	0.01	0.5	0.06

**Figure 6: Cost of different join operators.**

Due to relatively small data sizes, the gap between the adaptive join and the block nested loops join tuned using conservative selectivity estimates (i.e., $\sigma = 1$) is smaller. The adaptive join is up to 30% cheaper than the block join, while it only incurs overheads of less than 3% in one scenario (“Reviews”). The latter scenario features the join predicate with the highest selectivity, meaning that the conservative assumptions on selectivity made by the non-adaptive block join are (almost) correct.

The cost differences between tuple and block joins are primarily due to a large gap in terms of the number of tokens read. The number of written tokens is distributed more evenly. In the Reviews scenario, the tuple join even produces fewer tuples than the other join algorithms. This is due to the fact that the block joins require several tokens per result tuple, whereas the tuple join produces one token for each pair of tuples. As the selectivity of the join predicate is high in the “Reviews” scenario, the tuple join gains a slight advantage in terms of the number of generated tokens.

Similar to processing fees, switching to the block join algorithms reduces execution time. For instance, generating a complete join

**Figure 7: Output quality of different join operators.**

result in the first scenario (“Emails”) takes 435 seconds when using the tuple join, compared to three seconds with the adaptive join algorithm. The embedding join incurs significantly lower costs than the other operators. This is due to the use of a cheaper model, generating embeddings, and to the low number of tokens read. The embedding join reads all input data only once and does not generate any output tokens.

LOTUS consumes a similar number of tokens as the tuple nested loops join algorithm. Therefore, execution costs are comparable as well and significantly higher than for the block-based join algorithms. On the other hand, LOTUS is significantly faster than the tuple nested loops algorithm. Compared to the adaptive join algorithm, LOTUS is faster in one scenario (166 versus 31 seconds), while achieving comparable execution time in another (six versus five seconds), and increasing execution time for the Emails scenario (three versus 13 seconds). Clearly, the relative performance in terms of execution time is not aligned with the relative performance in terms of the number of tokens processed. This can be explained by the fact that LOTUS parallelizes LLM invocations, whereas the implementation of the join operators proposed in this paper is sequential. While the focus of the proposed implementations is on costs, rather than run time, different blocks of input tuples could be processed in parallel as well.

Figure 7 reports on the accuracy of different join operators. Specifically, it reports recall, precision, and the F1 score, measured by comparing the result tuples generated by different join operators to the ground truth result. In two of the three scenarios, using block joins, rather than the tuple join, leads to a slight degradation of F1 scores. However, in the first scenario, using the adaptive join over the tuple join almost doubles the F1 score. It seems that GPT-4 is able to identify pairs of contradicting statements better when seeing a larger sample of all available statements. This shows that, despite reducing costs and time by orders of magnitude, using block joins over the simple tuple join does not degrade result quality in general.

The embedding join has an F1 score of zero (with *both*, precision and recall, evaluating to zero) for the Email benchmark and an F1

score of 0.06 for the reviews. On the other hand, it has a perfect F1 score of one for Ads. This can be explained by the properties of the join predicates. For Ads, the goal is to find matching ads and searches. Here, having similar embedding vectors is indeed a good indicator for whether or not two tuples satisfy the join condition. For the Emails benchmark, on the other hand, the goal is to find contradicting statements. Such statements likely have dissimilar embedding vectors.

LOTUS achieves an optimal F1 Score in two out of the three scenarios. Interestingly, the F1 Score for the Emails scenario is significantly below the block-based join algorithms. This correlates with the result quality of the embedding-based join algorithm. LOTUS uses embeddings to speed up joins as well. Hence, this scenario, aimed at finding contradicting text, appears to be hard for embedding-based methods in general.

8 RELATED WORK

This work relates most to several recently proposed systems for semantic query processing [10, 11, 15, 16, 25, 36], 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 7 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 [29]. In contrast to joins, the aforementioned paper focuses on efficient implementations of semantic sort operators.

Join algorithms have been the focus of intensive research in the database community for many decades [30]. The join operators proposed in this paper are variants of nested loop joins, the most popular join operator for theta joins in general. However, the focus on language models implies several unique constraints, influencing not only the operator implementations but also the associated cost models and, therefore, the optimal settings for parameters such as tuple batch sizes. First, using simple, traditional cost models (based on the number of pages read and written), nested loop join variants require only one single output buffer page, independently of the join result size. This means that join selectivity does not influence optimal batch sizes for the input tables. Instead, for language models, the number of output tuples influences the number of tokens available for reading input. Second, traditional block nested loop join variants assume that we can load additional data into an input buffer while maintaining the content of other input buffers at no additional costs. Instead, language models incur costs for reading all relevant input tokens repeatedly, independently of whether the content changed, compared to the last invocation, or not. Therefore, maximizing the size of one input buffer while minimizing the size of the other, a strategy that works best for block nested loops join in a traditional setting, does not maximize performance when executing joins via language models (e.g., this becomes apparent in Figure 3).

This work connects to prior work that exploits language models for data management tasks [2, 6, 14, 21, 28, 32–34]. In particular, it connects to prior work leveraging language models for join processing [31]. 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 7). 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 [28], 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 [24], implementing relational operators with language models connects to prior work leveraging crowdsourcing for data processing [8, 18, 22, 23]. In particular, it connects to prior work leveraging human crowd workers for joins and related matching tasks [7, 17, 19, 37, 38]. 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, 4], GPUs [13, 39], and FPGAs [9]. The approaches presented in this paper target a different platform (namely: language models) with unique properties.

The work presented here also differs from recent work, leveraging machine learning to speed up traditional, relational processing [27]. Instead, this paper aims to expand the scope of relational processing via language models. The adaptive join algorithm connects to a rich body of work on adaptive query processing [3, 5, 12, 35]. However, the adaptive algorithm presented here aims at solving specific challenges that arise in the context of language models, in particular, the need to balance the input size with the expected output size.

9 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.

REFERENCES

- [1] Martina Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multicore database systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1064–1075. <https://doi.org/10.14778/2336664.2336678> arXiv:1207.0145
- [2] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Re. 2023. Language Models Enable Simple Systems for Generating Structured Views of Heterogeneous Data Lakes. *PVLDB* 17, 2 (2023), 92 – 105. <https://doi.org/10.14778/3626292.3626294>
- [3] Ron Avnur and Joe Hellerstein. 2000. Eddies: continuously adaptive query processing. In *SIGMOD*. 261–272. <https://doi.org/10.1145/342009.335420>
- [4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *Proceedings - International Conference on Data Engineering* (2013), 362–373. <https://doi.org/10.1109/ICDE.2013.6544839>

- [5] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: robust access path selection without cardinality estimation. *VLDB Journal* 27, 4 (2018), 521–545. <https://doi.org/10.1007/s00778-018-0507-8>
- [6] Zui Chen, Ju Fan, Sam Madden, and Nan Tang. 2023. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes. In *CIDR*. 1–7.
- [7] Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2012. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web* (2012), 469–478. <https://doi.org/10.1145/2187836.2187900>
- [8] MJ Franklin and D Kossmann. 2011. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*. 61–72. http://www.cs.berkeley.edu/~simrxin/papers/crowddb_sigmod2011.pdf
- [9] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoenes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. 2013. Accelerating join operation for relational databases with FPGAs. *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013* (2013), 17–20. <https://doi.org/10.1109/FCCM.2013.17>
- [10] Saehan Jo and Immanuel Trummer. 2023. Demonstration of ThalamusDB: Answering Complex SQL Queries with Natural Language Predicates on Multi-Modal Data. In *SIGMOD*. 179–182. <https://doi.org/10.1145/3555041.3589730>
- [11] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *SIGMOD* 2, 3 (2024), 1–26. <https://doi.org/10.1145/3654989>
- [12] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *CoRR abs/1802.0* (2018). [arXiv:1802.09180](https://arxiv.org/abs/1802.09180) <http://arxiv.org/abs/1802.09180>
- [13] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. *8th International Workshop on Data Management on New Hardware, DaMoN 2012 - In Conjunction with ACM SIGMOD/PODS Conference* (2012), 55–62. <https://doi.org/10.1145/2236584.2236592>
- [14] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2023. CHORUS: Foundation Models for Unified Data Discovery and Exploration. *CoRR abs/2306.0* (2023). [arXiv:2306.09610](https://arxiv.org/abs/2306.09610) <http://arxiv.org/abs/2306.09610>
- [15] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpsest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*. <https://github.com/mitdbg/palimpsest>
- [16] Samuel Madden, Michael Cafarella, Michael Franklin, and Tim Kraska. 2024. Databases Unbound: Querying All of the World's Bytes with AI. *PVLDB* 17, 12 (2024), 4564–4554. <https://doi.org/10.14778/3685800.368591>
- [17] Adam Marcus, Eugene Wu, and David Karger. 2011. Human-powered sorts and joins. In *VLDB*. 13–24. <http://dl.acm.org/citation.cfm?id=2047487>
- [18] Adam Marcus, Eugene Wu, DR Karger, Samuel Madden, and RC Miller. 2011. Crowdsourced databases: Query processing with people. In *CIDR*. 211–214. <http://dspace.mit.edu/handle/1721.1/62827>
- [19] Adam Marcus, Eugene Wu, David R Karger, Samuel Madden, Robert C Miller, Sigmod Acn, and New York. 2011. Demonstration of Qurk : A query processor for human operators. In *SIGMOD*. 1315–1318.
- [20] Priti Mishra and Margaret H. Eich. 1992. Join processing in relational databases. *ACM Computing Surveys (CSUR)* 24, 1 (1992), 63–113. <https://doi.org/10.1145/128762.128764>
- [21] Avnika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *PVLDB* 16, 4 (2022), 738–746. [arXiv:2205.09911](https://arxiv.org/abs/2205.09911) <http://arxiv.org/abs/2205.09911>
- [22] Aditya G. Parameswaran. 2013. Human-Powered Data Management. (2013), 225.
- [23] Aditya Ganesh Parameswaran, Hyunjung Park, Hector Garcia-Molina, Jennifer Widom, and Neoklis Polyzotis. 2012. Deco: declarative crowdsourcing. In *Information and Knowledge Management*. 1203–1212. <https://doi.org/10.1145/2396761.2398421>
- [24] Aditya G. Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2023. Revisiting Prompt Engineering via Declarative Crowdsourcing. (2023). [arXiv:2308.03854](https://arxiv.org/abs/2308.03854) <http://arxiv.org/abs/2308.03854>
- [25] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. In *Proceedings of the VLDB Endowment*, Vol. 18. 4171–4184. <https://doi.org/10.14778/3749646.3749685>
- [26] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill New York. 1–1104 pages.
- [27] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *PVLDB* 16, 7 (2023), 1749–1762. [arXiv:2111.08824](https://arxiv.org/abs/2111.08824) <http://arxiv.org/abs/2111.08824>
- [28] Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying Large Language Models with SQL. *CoRR abs/2304.0* (2023). [arXiv:2304.00472](https://arxiv.org/abs/2304.00472) <http://arxiv.org/abs/2304.00472>
- [29] Fuheng Shao, Jiayue Chen, Yiming Pan, Tahseen Rabbani, Divyakant Agrawal, and Amr El Abbadi. 2025. Access Paths for Efficient Ordering with Large Language Models. *CoRR 2509.00303* (2025). <https://arxiv.org/abs/2509.00303>
- [30] Leonard D. Shapiro. 1986. Join processing in database systems with large main memories. *ACM Transactions on Database Systems* 11, 3 (1986), 239–264. <https://doi.org/10.1145/6314.6315>
- [31] Sahaana Suri, Ihab Ilyas, Christopher Re, and Theodoros Rekatsinas. 2021. Ember: No-Code Context Enrichment via similarity-based keyless joins. *PVLDB* 15, 3 (2021), 699–712. [arXiv:2106.01501v1](https://arxiv.org/abs/2106.01501v1)
- [32] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Halevy. 2021. From natural language processing to neural databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1033–1039. <https://doi.org/10.14778/3447689.3447706>
- [33] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *PVLDB* 15, 11 (2022), 2921 – 2928. <https://doi.org/10.14778/3551793.3551841>
- [34] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that “Reads the Manual”. In *SIGMOD*. 190–203. <https://doi.org/10.1145/3514221.3517843>
- [35] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Transactions on Database Systems* 46, 3 (2021), 1–45. <https://doi.org/10.1145/3464389>
- [36] Matthias Urban and Carsten Binnig. 2024. CAESURA: Language Models as Multi-Modal Query Planners. In *CIDR*.
- [37] Weijia Wang and Michèle Sebag. 2013. Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search. *Machine Learning* 92, 2-3 (2013), 403–429. <https://doi.org/10.1007/s10994-013-5369-0>
- [38] Steven Euijong Whang, Peter Lofgren, and Hector Garcia-Molina. 2013. Question selection for crowd entity resolution. In *Proceedings of the VLDB Endowment*, Vol. 6. 349–360. <https://doi.org/10.14778/2536336.2536337>
- [39] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The yin and yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828. <https://doi.org/10.14778/2536206.2536210>