CEDAR: A System for Cost-Efficient Data-Driven Claim Verification

Tharushi Jayasekara
tkj27@cornell.edu
Cornell University
Ithaca, NY, USA

Immanuel Trummer
itrummer@cornell.edu
Cornell University
Ithaca, NY, USA

ABSTRACT

We present CEDAR, a system for cost-efficient, data-driven claim verification. CEDAR takes as input a collection of text documents, containing claims that can be verified from relational data. The system uses large language models (LLMs) to map claims to SQL queries that can be used for claim verification. While LLMs like GPT-4 are nowadays able to map claims to queries with high accuracy, using them is expensive. This is why CEDAR implements multiple verification approaches, ranging from zero-shot LLM invocations to iterative, agent-based approaches, that realize different tradeoffs between accuracy and costs. The system may apply multiple methods to the same claim, starting with cheaper methods and resorting to more expensive versions in case of failures. CEDAR uses cost-based optimization to derive an optimal order of verification methods and an optimal number of re-tries (with randomization) for each method, enabling users to trade costs for accuracy via tuning parameters. The experiments on real data, including newspaper and Wikipedia articles, show that CEDAR achieves significantly higher accuracy than prior methods for data-driven fact-checking.

PVLDB Reference Format:
Tharushi Jayasekara and Immanuel Trummer. CEDAR: A System for Cost-Efficient Data-Driven Claim Verification. PVLDB, 18(11): 4492 - 4504, 2025.
doi:10.14778/3749646.3749708

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TharushiJay/CEDAR.

1 INTRODUCTION

Data from large relational databases are most effectively communicated through textual summaries that highlight key statistics. Data-centric roles that make use of such summarizations range from public health analysts who analyze disease incidence to data journalists who analyze data for insights to be published in newspaper articles. Improperly analyzed data could cause repercussions across many industries such as scientific paper retractions in academia, financial losses in retail, flawed regulatory measures in policy-making as well as environmental damage in energy production. Our primary goal is to assist authors of such summaries in identifying factual inaccuracies, much like how a spell checker flags spelling errors.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749708

We present CEDAR, a tool that supports authors and readers of data summaries to verify claims about relational data. CEDAR is applied to collections of text documents containing claims that refer to associated data sets. Internally, CEDAR translates claims into SQL queries. Executing those queries and comparing the query result to the claimed result enables the system to mark up claims as correct or incorrect. On the surface, this problem may resemble the classical problem of text-to-SQL translation. However, instead of translating single questions, we translate claims embedded in larger text documents that come with a claimed result. Those differences motivate a specialized system design, e.g., featuring a multi-stage verification approach that exploits claimed values as a signal to assess the likelihood of a correct query translation.

Example 1.1. Consider the claim "The two fatal accidents involving Malaysia Airlines this year were the first for the carrier since 1995." extracted from a newspaper article on 538 [3]. This claim is derived from a data set, available on the 538 Web site. Our goal is to verify the claim value 'two' by generating an SQL statement that queries for the number of fatal accidents involving Malaysia Airlines on the aforementioned data, stored as table `airlines`. This claim maps to the SQL query `SELECT "fatal_accidents_00_14" FROM airlines WHERE airline = 'Malaysia Airlines'`. By executing this query and comparing its result with the claim value, we can verify whether the claim is *correct* or *incorrect*.

CEDAR exploits agents, backed by state-of-the-art LLMs like GPT-4, to translate claims into SQL queries. Agents are a recently proposed method that leverages LLMs in an iterative approach to solve complex problems. The LLM is used to decompose the problem, as well as to invoke functions, so-called tools, to solve sub-problems. In the case of data-driven claim verification, tools give the agent access to the target data and compare candidate SQL queries to claim values. Special care must be taken to avoid scenarios in which the agent can "cheat", i.e., seemingly solve a given task but without producing a useful solution. For instance, providing the agent with access to the claim value may often result in SQL queries that merely return a constant value (the claim value) but do not accurately represent the semantics of the input claim. Therefore, CEDAR takes great care to obfuscate key information for the agent while providing it with enough context to translate the claim into an SQL query. Furthermore, as agents may collect relevant information for claim verification across multiple iterations, CEDAR must perform a post-processing step in which it analyzes traces of tools invocations, created by the agent, to compose a single SQL query translating the input claim.

Agents are a powerful method that is typically able to map even complex claims to the appropriate queries. However, the iterative approach increases the amount of text read and generated by the

LLM. As LLM processing fees are typically proportional to the amount of text, verifying claims via agents incurs high verification costs per claim. CEDAR reduces monetary processing fees by a multi-stage verification approach. Instead of applying the most powerful and expensive methods first (i.e., agents), it starts by applying cheaper verification methods that are in many cases sufficient. CEDAR features multiple (non-iterative) single-shot claim-to-SQL translation methods, requiring only a single invocation to the LLM. To boost the chances of a successful translation, CEDAR automatically collects samples of correctly translated claims and uses them as samples for few-shot learning. This approach includes samples of successfully solved problem instances into the prompt text, sent to the LLM. If the results of queries, translated via cheaper methods, are far off the claimed values, CEDAR may decide to apply more expensive verification methods to the same claim.

Each of the aforementioned verification methods can be used with a variety of language models. Altogether, this leads to a large space of possible verification methods, realizing different tradeoffs between success probability (i.e., the chances to accurately translate the claim to an SQL query) and monetary verification fees. As the output of LLMs is randomized (i.e., the LLM may generate different answers in multiple invocations with the same input), it is even reasonable to re-try verification on the same claim with the same approach. CEDAR exploits data obtained via profiling different verification approaches on samples of the input data. Given user preferences on the desired cost-quality tradeoff, it determines an optimized verification schedule. This schedule specifies which verification methods to use and how many retries to apply before switching to the next method.

CEDAR uses cost-based optimization to determine the schedule to use. The problem of finding an optimal verification schedule relates to the problem of optimally ordering expensive predicates (taking into account their selectivity and costs). However, it differs by the possibility of retrying verification methods with possibly different outcomes. We prove that our scenario-specific cost function satisfies the principle of optimality, and present a dynamic programming algorithm, generating an optimal schedule according to our cost model.

In summary, our original, scientific contributions are as follows:

- We introduce CEDAR, a system for cost-efficient, data-driven fact-checking.
- We present multiple claim verification methods, translating claims to SQL queries using LLMs.
- We describe a framework for multi-stage claim verification and associated optimization methods.
- We report on the results of experiments, evaluating cost and accuracy of CEDAR in comparison to baselines.

The remainder of this paper is organized as follows. Section 2 defines the problem, while Section 3 gives an overview of the CEDAR system. Section 4 describes CEDAR's multi-stage verification approach. Section 5 describes several approaches for claim verification, based on LLMs. Section 6 describes how we generate optimized schedules for multi-stage claim verification. Section 7 reports experimental results, comparing CEDAR to multiple baselines. Section 8 describes prior work that relates to CEDAR before we conclude in Section 9.

2 PROBLEM STATEMENT

We introduce our formal model.

Definition 2.1. A text **Document** contains claims summarizing a data set. Given document d, we denote by $d.claims$ the claims in the document text and by $d.data$ a relational database that the claims refer to.

Definition 2.2. A **Claim** is defined by a sentence and the position of a numeric or textual value within that sentence. This value represents either an aggregate value derived from the data set associated with the document containing the claim, or a specific data entry extracted based on filtering conditions. Furthermore, a claim may be associated with additional text, providing relevant context to interpret the claim sentence. Given a claim c, we denote by $c.sentence$ the claim sentence and by $c.span$ the position of the claim value within that sentence, where $c.span.start$ and $c.span.end$ denote the start and end positions. Also, we denote by $c.context$ any relevant context. Finally, we denote the claim value by $c.value$.

Example 2.3. Consider the claim in Example 1.1. Here, $c.sentence$ has the value "The two fatal accidents involving Malaysia Airlines this year were the first for the carrier since 1995.", $c.span.start = 1$ and $c.span.end = 1$ since the claimed value appears at index 1. $c.context$ is the paragraph containing the claim sentence.

Definition 2.4. A **Correct Claim** is a claim for which the claim value corresponds to the result of executing a query that represents the claim semantics. Our approach supports arbitrary SQL queries that return a single cell as output, either a scalar number or a textual (string) value. Given a claim c, we denote by $c.value$ the associated claim value (which appears in the claim sentence at $c.span$). We denote by $c.query$ an SQL query that translates the claim semantics. If the result of executing $c.query$ on the data associated with the document is equivalent to $c.value$, the claim is correct. Otherwise, the claim is incorrect. Note that we consider a claim as correct if, for numeric claim values, the query result can be rounded to the claimed value (since claims in text often round results) and, for textual claims, the query result exactly matches the claimed value.

Example 2.5. For the claim c from the previous example, it is $c.query$ =`SELECT "fatal_accidents_00_14" FROM airlines WHERE airline = 'Malaysia Airlines'` and $c.value = 2$. Since the result of the query is equivalent to the claimed value, the claim is correct. Note that a query result of, e.g., 2.1 could be rounded to two and is therefore also equivalent to the claimed value.

CEDAR solves the problem defined next.

Definition 2.6. Given a set of documents D as input, the goal of **Claim Verification** is to map the claims $d.claims$ of each document $d \in D$ to a correctness flag ($c.correct$) and an SQL query ($c.query$) that represents the claim semantics.

3 SYSTEM OVERVIEW

Figure 1 shows an overview of the CEDAR system. CEDAR's input is a set of text documents to verify, along with associated data sets. Additionally, users can tune a parameter that enables them to trade verification precision for monetary processing fees. Setting a higher accuracy threshold will encourage CEDAR to verify more

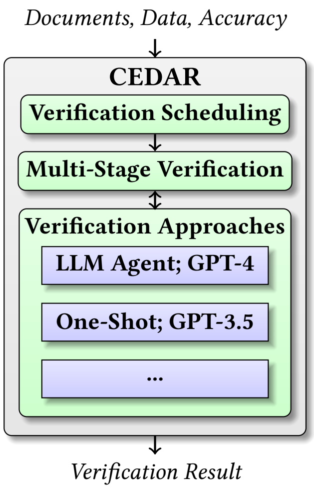

Figure 1: Architecture of CEDAR

thoroughly, using more expensive verification methods. The output of CEDAR is a verification result, mapping claims that appear in the input documents to verification results (i.e., whether the claim is correct or incorrect), as well as to SQL queries that were used for verification.

Internally, CEDAR first determines the order in which different verification approaches are applied, as well as the number of retries for each approach. CEDAR applies cost-based optimization to find an optimal verification schedule. For that, it exploits profiling statistics, providing estimates for the success probability and the average costs of different verification approaches. Furthermore, CEDAR takes into account user preferences on the cost-quality tradeoff (the accuracy threshold specified as input). If users choose a lower accuracy, CEDAR has more options to reduce verification costs by choosing low-cost verification variants, even if it reduces the quality of the verification result.

After selecting a verification schedule, CEDAR performs multi-stage verification. This means that CEDAR applies different verification approaches with a given number of retries until it seems likely that claims have been mapped accurately to SQL queries that can be used for verification. If a verification approach fails, CEDAR proceeds to the next approach, according to the previously chosen order. CEDAR features a variety of verification methods that realize a wide range of cost-quality tradeoffs. Those approaches include agent-based, iterative verification methods as well as (cheaper) one-shot verification methods. All of those approaches use large language models for claim-to-SQL translation (the current implementation relies on OpenAI's GPT model series). CEDAR instantiates approaches with different model variants, expanding the range of cost-quality tradeoffs CEDAR is able to choose from.

4 MULTI-STAGE CLAIM VERIFICATION

Algorithm 1 describes multi-stage claim verification, as executed by CEDAR. CEDAR takes as input a collection of documents to verify, profiling results describing different claim verification methods, and an accuracy threshold. The latter threshold enables users to trade

Algorithm 1 Multi-Stage Claim Verification

```
// Given a list of documents, profiling data, and an accuracy
// constraint, verify the claims in the documents
function CEDAR(docs, profiling_data, acc_constraint)
    // Generate optimal verification schedule
    schedule ← SCHEDULE(profiling_data, acc_constraint)
    // Iterate over documents to verify
    for d ∈ docs do
        // Retrieve all claims to verify
        claims ← d.claims
        // Iterate over verification schemes
        for vScheme ∈ schedule do
            // Try verification without sample
            S ←VERIFY(vScheme, claims, ⟨⟩, d.data)
            // Remove successfully verified claims
            claims ← claims \ S
            // Did we collect at least one sample?
            if S ≠ ∅ then
                // Retry verification with sample
                sample ← element from S
                S ←VERIFY(vScheme, claims, sample, d.data)
                // Remove successfully verified claims
                claims ← claims \ S
            end if
        end for
    end for
    // Return documents with verification results
    return docs
end function
```

accuracy for computation fees. Setting a lower threshold enables the system to select cheaper (and less accurate) verification methods, thereby saving costs. The system takes accuracy targets as input, rather than a cost budget. CEDAR selects verification methods that meet accuracy targets (under several simplifying assumptions) while minimizing costs. As output, Algorithm 1 returns documents whose claims are annotated with verification results.

First, CEDAR uses cost-based scheduling to determine an optimal verification schedule. The verification schedule determines the order in which different verification approaches are tried and the number of re-tries to execute with each approach. The details of the scheduling algorithm are discussed in Section 6.

Next, CEDAR iterates over all documents to verify. Each document d is associated with a set of claims ($d.claims$) to verify. At each step, Variable *claims* contains the remaining claims to verify (which is initialized to $d.claims$). For the current document, CEDAR iterates over all verification approaches in the order that was scheduled before. The success probability for verification approaches increases if document and approach-specific samples (of claims with associated SQL queries) are available for few-shot learning. Initially, no such samples are available. CEDAR tries verifying the current claims without samples, using the VERIFY function. This function applies a single verification approach to all claims. Its pseudo-code (Algorithm 2) is discussed later in this section.

If at least one claim was successfully verified ($S \neq \emptyset$), CEDAR retries verification with the current method on the remaining claims.

Algorithm 2 Verifying Claims with Given Method

```
// Verify a given list of claims using a translated sample and a
// database containing data about the claims
function VERIFY(vScheme, claims, sample, database)
    // Initialize successfully verified claims
    S ← ∅
    // Iterate over claims to verify
    for c ∈ claims do
        // Retrieve masked claim and context
        ⟨m, ctx⟩ ← PRE_PROC(c.para, c.sent, c.span)
        // Invoke the verification scheme on the claim
        q ← vSCHEME(m, database, sample, ctx)
        // Attach query to claim
        c.query ← q
        // Is query likely to be correct?
        if CORRECTQUERY(q, c.result, database) then
            // Determine correctness based on query
            c.correct ←CORRECTCLAIM(q, c.result, database)
            // No prior sample available?
            if sample = ⟨⟩ then
                return {c}
            end if
            // Add to successfully verified claims
            S ← S ∪ {c}
        end if
    end for
    return S
end function
```

After each invocation of VERIFY, the successfully verified claims are removed from the set of remaining claims.

Algorithm 2 realizes one single verification stage, iterating over all input claims using a given claim verification method. For each claim, CEDAR pre-processes the claim text (discussed in more detail in the next section), and invokes the current verification method which results in an SQL query. If claim translation succeeds, this SQL query represents the semantics of the input claim. Prior research [17] shows that incorrect, numerical claims typically use numbers that are relatively close to accurate values. This principle has been exploited by prior work on data-driven fact-checking as well [14]. CEDAR leverages this insight to verify whether the queries returned by the claim translation method are plausible, i.e., whether their result is close enough to the claimed value to be likely correct. For that, Algorithm 2 uses Function CORRECTQUERY (pseudo-code omitted due to space restrictions) to execute the query on the input data and compare the result to the claimed value. For numerical claims, a query is deemed plausible if its result is in the same order of magnitude as the claim value. For textual claims, we compute the similarity between the embedding vectors of claimed and retrieved values using the model 'MiniLM-L6' [33]. If the similarity exceeds a particular threshold, the query is considered plausible. We set the similarity threshold to 0.7, which indicates moderate-to-strong semantic alignment between short text spans and accounts for errors due to abbreviations and spelling mistakes. Function CORRECTQUERY returns true when the query produced by the claim verification method is assumed to be likely correct.

Algorithm 3 Validating Claims

```
// Given a query, database, and the claim result, validate
// the correctness of the claim
function CORRECTCLAIM(query, c_res, data)
    // Execute query on target database
    q_res ← EXECUTEQUERY(q, data)
    if ISNUMERIC(c_res) then
        // Get precision of claim value
        precision ←GETPRECISION(c_res)
        // Round query result to claim precision
        rounded_q_res ← ROUND(q_res, precision)
        // Check whether rounded query result matches claim
        return c_res = rounded_q_res
    else
        // Convert to dense vector representations
        c_vec ← GETEMBEDDING(c_res)
        q_vec ← GETEMBEDDING(q_res)
        // Measure semantic overlap
        sim ← COSINESIMILARITY(q_vec, c_vec)
        return c_res ≥ 0.8
    end if
end function
```

If the translated query is likely to be correct, CEDAR assigns a correctness value to the current claim determined by Function CORRECTCLAIM, specified as Algorithm 3. This function executes the query on the corresponding data. For numerical claims, it determines the precision of the claim value, rounds the query result to that precision, and compares the resulting numbers. If the rounded query result matches the claim, the claim is correct (otherwise incorrect). For textual claims, if the similarity exceeds 0.8, the claim is correct (otherwise incorrect). If none of the approaches succeed in generating an executable query, the claim is assumed to be unverifiable from the data and marked as correct by default. If executable queries are generated but no query result matches the claimed value, the claim is marked as incorrect.

Example 4.1. A query result of 3.140 matches a claimed value of 3.1 and a claimed value of 3 but not a claimed value of 3.143. On the other hand, a query result of 3.143 matches a claimed value of 3.14.

For (likely) correct queries, Algorithm 2 checks for the special case that the current claim is the first one that was successfully translated. In that case, the input sample is the empty tuple ($sample = \langle\rangle$) and Algorithm 2 returns immediately the single, successfully translated claim. This is done to enable the calling function (Algorithm 1) to use the successfully verified claim as a sample for the next invocation of VERIFY (since adding samples of successfully verified claims tends to increase the performance of LLM-based verification methods). If a sample is already available, Algorithm 2 adds the current claim to the set of successfully verified claims that is ultimately returned as result.

Correct:

```sql
SELECT fatal_accidents_00_14
FROM airlinesafety
WHERE airline = 'Malaysia Airlines';
```

Incorrect:

```sql
SELECT 2 AS fatal_accidents
FROM airlinesafety
WHERE airline = 'Malaysia Airlines'
  AND fatal_accidents_00_14 = 2;
```

Figure 2: The effect of masking the claim value

Algorithm 4 Preprocessing Claim Text

// Given the paragraph, context, and position of the claim value,
// generate the masked claim and context information
function PRE_PROC($para, sent, span$)
  // Obfuscate the claim value in the sentence
  $masked\_claim \leftarrow sent.replace(span, "x")$
  // Obfuscate the sentence in the paragraph
  $context \leftarrow para.replace(sent.span, masked\_claim)$
  // Return masked claim and context
  return $\langle masked\_claim, context \rangle$
end function

5 VERIFICATION APPROACHES

We discuss pre-processing of claim text and context in Section 5.1. In Section 5.2, we discuss cheap translation methods that involve only one single invocation of the LLM. Next, in Section 5.3, we discuss more complex methods, based on LLM agents. Using agents requires a post-processing stage in which we compose SQL queries representing claims in their entirety from multiple SQL queries, issued by the agent over the course of the claim verification process.

5.1 Pre-Processing

During pre-processing, we obfuscate certain parts of the claim text and context. The following example illustrates the reason.

Example 5.1. Consider the claim "The **2** fatal accidents involving Malaysia Airlines this year were the first for the carrier since 1995", taken from a 538 article. If prompting the LLM to create an SQL query for claim verification, providing unchanged text as input, the LLM generates the query shown on the right side of Figure 2. This query contains the claimed result as a constant. On the other hand, if masking the claim value appropriately (as discussed next), the LLM generates the correct query shown on the left.

To avoid cases like the one depicted in Example 5.1, CEDAR obfuscates claim values in the claim text and related context. Algorithm 4, invoked in Algorithm 1, takes as input context (the text paragraph containing the claim), the claim sentence, and the position within the claim sentence at which the claim value appears. Next, Algorithm 4 obfuscates the claimed value in the claim sentence and, finally, replaces the original claim sentence with the changed version in the surrounding paragraph as well. The resulting text avoids the LLM taking "shortcuts" by creating SQL queries that contain the claim value as a constant.

5.2 One-Shot Translation

CEDAR supports cheap one-shot verification. Here, the LLM is called only once to translate the claim into an equivalent SQL query.

Algorithm 5 One-shot Claim-to-SQL Translation

// Invoke one-shot translation approach using masked claim,
// data, a translated sample, and context information.
function ONESHOTVERIFICATION($m, data, sample, ctx$)
  // Generate the prompt
  $prompt \leftarrow$ ONESHOTPROMPT($m, data, sample, ctx$)
  // Invoke the language model using the prompt
  $response \leftarrow$ INVOKELLM($prompt$)
  // Extract the SQL query from the response
  $q \leftarrow$ EXTRACTQUERY($response$)
  // Return translated query
  return q
end function

```
Given the claim "{claim}" where "x" is a "{type}"
value, you must think about a question that
generates "x" as the answer and then generate a SQL
query to answer that question.

You must use the schema of the following table
called "table". {db_schema}

To query for percentages use the format
"SELECT (SELECT COUNT(column_name) FROM table
WHERE equality_predicates)
* 100.0/ (SELECT COUNT(column_name) FROM table
WHERE equality_predicates)".
Other queries are of format
"SELECT aggregate_function(column_name) FROM table
WHERE equality_predicates".

Wrap the SQL in ```sql ```.
{sample}
The following context information might help
to form the SQL query. {context}
```

Figure 3: Prompt template for one-shot verification

Algorithm 5 describes the steps for one-shot verification. First, CEDAR generates a prompt that describes the claim to translate and relevant context (including information about the structure of the data the claim refers to and the claim context in the input document). Next, this prompt is sent to the LLM which generates a response text. If claim translation succeeds, that response contains an SQL query (marked up according to the specifications in the prompt). Algorithm 5 finally extracts and returns that query.

Figure 3 shows the prompt template for one-shot claim translation. Placeholders are marked via curly braces (e.g., `{claim}`). First, the prompt template describes the claim (with obfuscated claim value), along with general instructions on the translation task. Second, the prompt contains the data type of the claim value (placeholder `{type}`). We use 'numeric' to denote numeric claims, and leave this placeholder empty for others to allow for generalizability. Third, the template describes the schema of the database from which the claim can be verified (placeholder `{db_schema}`). Next, the prompt contains high-level suggestions about the types of queries to consider (the instructions are inspired by the query search space explored by the AggChecker system [14], whose benchmark we use in the experiments). Note that the LLM is free to explore (and, in practice does explore) queries that do not comply with the suggested format, too. Fourth, the template instructs the LLM to mark up the generated SQL query in the answer, making it easier to extract it during post-processing. If available, a sample of a correctly translated claim is included in the prompt for few-shot learning (`{sample}`). Finally, the prompt contains relevant context, extracted from the text document. While providing the entire text as input is possible, doing so would increase the number of tokens read (and their verification costs) significantly. On the other hand, some amount of context can be required to accurately translate claims. As a compromise, we choose to provide the LLM with the paragraph containing the claim sentence as input (`context`).

Table 1: Placeholder values for a sample prompt

<table>
<tr><td>Claim</td><td>The first Grand Prix winner was ‘x’ at the 1950 British Grand Prix.</td></tr>
<tr><td>Type</td><td>“"</td></tr>
<tr><td>DB Schema</td><td>
<table>
<tr><td>Country</td><td>Driver</td><td>...</td><td>Wins</td></tr>
<tr><td>United Kingdom ...</td><td>Lewis</td><td>...</td><td>105</td></tr>
<tr><td>Germany ...</td><td>Michael</td><td>...</td><td>91</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
</table>
</td></tr>
<tr><td>Sample</td><td>For example, given the claim “‘x’ holds the record for the most race wins in Formula One history, with 105 wins to date.", to find the value for “x", generated SQL query would be “SELECT “Driver" FROM table WHERE “Wins" = (SELECT MAX(“Wins") FROM table)".</td></tr>
<tr><td>Context</td><td>As of the 2025 Spanish Grand Prix, out of the 781 drivers who started a Grand Prix,[14] there have been 115 Formula One Grand Prix winners. ...</td></tr>
</table>

Example 5.2. Table 1 shows how placeholders in the template from Figure 3 are substituted for an example claim.

Algorithm 6 Agent-Based Verification Approach

```
// Invoke the agent-based verification approach using
// masked claim, data, translated sample, and context.
function VerificationByAgent(m, data, sample, ctx)
    // Generate prompt for agent
    prompt ← AgentPrompt(m, data, sample, ctx)
    // Invoke iterative agent approach
    query_list ← ReActAgent(prompt)
    // Reconstruct query via post-processing
    q ← QueryReconstruction(query_list, data)
    // Return reconstructed query
    return q
end function
```

5.3 Agent Approach

The one-shot approach discussed previously fails for complex claims. Among other things, the LLM may not have enough information about the target database (e.g., the precise names of constants that appear in the data) to guarantee an accurate translation. However, apriori, it is hard to determine which information is needed for a specific claim. This motivates an iterative approach, giving the LLM multiple tries and the possibility to request additional information about the data, once it becomes clear that it is relevant.

CEDAR supports verification approaches based on ReAct [37]. This approach uses LLMs to solve complex tasks. The LLM is invoked iteratively for “reasoning” and “action”. Via reasoning, the LLM can decompose complex tasks into sub-problems. Via action, the LLM can decide to invoke one out of a given collection of functions (so-called “tools”) with task-specific input parameters. The result of the function invocation is shared with the LLM in the prompt of the next LLM invocation.

Algorithm 6 describes CEDAR’s agent-based verification approach. CEDAR first generates a prompt to describe the verification task to the language model. The prompt template is related to the one in Figure 3 but expands this prompt by a description of tools available to the agent (described later in this section), as well as additional instructions (e.g., related to output formats and problem decomposition) enabling the ReAct approach. For that, we use standard prompt templates offered by the LangChain framework [2]. Next, CEDAR executes iterative verification via the ReAct approach. The result of verification is not a single query but all SQL queries executed by the ReAct agent. This is needed since the ReAct agent may piece together relevant information for claim verification over multiple queries, resulting in trivial queries (containing constants obtained via prior queries) at the end. In a post-processing stage, explained in more detail in the following subsection, CEDAR composes query fragments into one complete SQL query representing claim semantics.

Algorithm 7 implements the ReAct approach and is invoked from Algorithm 6. Starting from the original prompt, the algorithm iterates until the LLM decides to stop iterations (the `thought` property is set to `[Finished]` in that case). In each iteration, the language model is prompted to generate a “thought”, describing the next steps to take in order to solve the input problem. Each thought may describe an action. An action is the invocation of a tool, recommended by the LLM. We use two tools for claim verification. The first tool enables the LLM to obtain information on the unique values of certain columns in the data set. This turns out to be crucial to formulate accurate SQL queries. Providing the LLM with information on all unique values in each column is very expensive. Hence, it is better to enable the LLM to “decide” itself, whether or not values in specific columns are relevant for the input claim. The second tool enables the LLM to “test” SQL queries on the input data. More precisely, it enables the LLM to compare the result of a single query to the claim value. It can use the results to guide its efforts in mapping the input claim to the correct SQL query. We omit the pseudo-code for the first tool while we describe the code for the second tool later. In Algorithm 7, we parse actions from LLM thoughts (mapping them to the correct tool and to the corresponding tool input parameters), execute the action suggested

Algorithm 7 Iterative ReAct Agent

```
// Invoke the 'thought/action/observation' process of the ReAct
// agent using the given prompt, return list of queries.
function ReActAgent(prompt)
    // Initialize list of queries
    Q ← []
    // Analyze the prompt to start the thought process
    thought ← agent.analyze(prompt)
    // Repeat the thought process until it reaches an answer
    while thought.answer ≠"[Finished]" do
        // Based on reasoning, decide if an action is required
        if thought.requires_action() then
            // Decide which action to take
            action ← thought.selectTool()
            // Generate action input
            input ← thought.generateInput( )
            // Execute the action based on generated input
            output ← action.execute(input)
            // Log queries if any
            if action = DatabaseQuerying then
                Q ← Q ∘ [input]
            end if
            // Analyze the output to continue reasoning
            observation ← agent.analyze(output)
            thought ← agent.analyze(observation)
        else
            // Continue reasoning based on its knowledge
            thought ← agent.analyze( )
        end if
    end while
    // Return list of queries
    return Q
end function
```

by the LLM, and feed the result of the invocation to the LLM for the next iteration.

If an action invokes the querying tool, allowing the LLM to issue SQL queries on the input data, we log the corresponding queries in a list (Q). We return the list of queries as the final result.

Algorithm 8 Database Querying Tool Algorithm

```
// Query data using query q and compare
// query result to claimed result.
function DatabaseQuerying(q, data, claim_res)
    // Execute query on input data
    query_res ← ExecuteQuery(q, data)
    //  Get feedback comparing query result to claimed result
    return getFeedback(query_res, claim_res)
end function
```

Algorithm 8 shows pseudo-code for the tool allowing the LLM to query the input data. In addition to the query result, the tool returns a short feedback message to the agent comparing the query result to the claim value using Function CorrectQuery discussed earlier. For numbers, the feedback distinguishes between 'close', 'greater' and 'smaller', whereas for embedding vectors it distinguishes between 'matched' and 'mismatched', based on a similarity threshold. Of course, we could provide the LLM directly with the claim value as input. However, this leads to the problem illustrated in Example 5.1. On the other hand, giving the LLM some information on whether or not the results of queries come close to the claimed value can provide the LLM with valuable guidance, allowing it to correct its candidate queries. We balance between the two extremes (no information and all information on the claim value) by providing the LLM agent with high-level, comparative information only. As the second component of the output tuple, Algorithm 8 returns a short text, comparing the query result to the claim value. This information is sufficiently imprecise to avoid scenarios in which the LLM creates queries that contain the claimed value as a constant. On the other hand, the information is precise enough to enable the LLM to correct mistakes in the initial query over multiple iterations.

Example 5.3. Figure 4 illustrates how the agent approach processes an example query. The output shows all agent thoughts, as well as tool invocations and results. The agent tries a first SQL query using the querying tool. However, the query result is empty, leading to an error message. This motivates the agent to obtain a list of unique values in one of the relevant columns. Indeed, it turns out that the query should use a different constant from the prior version (i.e., `USA` instead of `United States`). The agent corrects the constant and issues a new query, resulting in the claimed value.

5.4 Post-Processing

The agent approach enables the LLM to issue multiple queries on the test database. The final query often contains constants, obtained by issuing prior queries. Therefore, we need to add a post-processing stage, reconstructing the complete SQL query from the queries issued over all iterations of the agent.

Algorithm 9 obtains as input a list of SQL queries issued by the agent while verifying a single claim (together with the data the queries refer to). The output is one single query, obtained by substituting constants that appear in the last query in the input list with prior queries with matching results. Given a list with only a single element, the last remaining query is returned (which may have been subject to substitutions in prior instances of the recursive method). If more than a single query remains, Algorithm 9 executes that query on the input data. Next, the algorithm iterates over all other queries, searching for constants that match the query result. If so, it is assumed that the agent derived the constant from the prior query result. Note that we consider substitutions only in one direction, substituting queries for constants that appear in later queries. This is reasonable since the input list contains queries in the order in which they were issued by the agent (and the agent can only derive constants from queries executed previously). Algorithm 9 decomposes each query into terms split by a white space delimiter. For numerical query results, the numerical term with minimal absolute distance to the query result is substituted. If the query result can be rounded to that term (note that it may be possible to round the query result to multiple terms but we consider the one with minimal distance most likely), the algorithm substitutes the token with the query. For string results, it replaces the string literal

Thought: To find the value of "x" in the claim "The French consume more wine than people in any other country - 370 glasses of wine per person per year, compared to just "x" glasses in the U.S.", we need to determine the wine consumption per person in the U.S. from the given table.

The table schema provided includes the columns: "country", "beer_servings", "spirit_servings", "wine_servings", and "total_litres_of_pure_alcohol". We are interested in the "wine_servings" column for the country "United States".

Thought: The correct SQL query to find the wine consumption per person in the U.S. is:
``` sql
SELECT "wine_servings" FROM table WHERE "country" = 'United States'
```
Action: database_querying
Action Input: SELECT "wine_servings" FROM table WHERE "country" = 'United States'
index 0 is out of bounds for axis 0 with size 0

Observation: It appears that there might be an issue with the country name used in the query. The country name might be different in the dataset. To resolve this, I will first check the unique values in the "country" column to ensure the correct country name is used.
Action: unique_column_values
Action Input: country

```
   country
0  Barbados
1  Botswana
.
50  USA
.
.
```

Observation: The unique values in the "country" column indicate that the United States is represented as "USA". I will now modify the SQL query to use "USA" instead of "United States".
Action: database_querying
Action Input: SELECT "wine_servings" FROM table WHERE "country" = 'USA'
[84, 'Value is correct']

Thought: I now know the final answer.
Final Answer: 84

Figure 4: Example for agent-based approach

directly with the sub-query. Finally, the algorithm performs further substitutions via a recursive call.

6 OPTIMAL SCHEDULING

CEDAR uses a multi-stage verification approach. Using the claimed value as a signal to detect successful verification, CEDAR proceeds to the next verification approach whenever the last one fails for a limited number of re-tries. The order in which verification approaches are tried as well as the number of retries per approach

Algorithm 9 Query Reconstruction Algorithm

```
// Given a list of SQL queries, reconstruct a single query that
// generates the expected answer by recursively replacing its
// constants with sub-queries.
function RECONSTRUCT(query_list, data)
    // Retrieve and remove first query in list
    cur_query ← query_list.REMOVE(0)
    // Last remaining query is complete
    if query_list.length = 0 then
        return cur_query
    end if
    // Execute query on target data
    res ← EXECUTEQUERY(cur_query, data)
    // Iterate over all remaining queries
    for i ← 0..query_list.length do
        // Split query using white space as delimiter
        query ← query_list[i]
        query_parts ← SPLIT(query)
        if ISNUMERIC(res) then
            // Identify minimum-distance part
            m ← arg min_{t∈query_parts:t∈ℝ} |t − res|
            // Check whether query part rounds to result
            if t rounds to res then
                // Replace part with sub-query
                n ← query.REPLACE(t, cur_query)
                // Replace old query in list
                query_list[i] ← n
            end if
        else
            // Replace with sub-query
            n ← query.REPLACE(res, cur_query)
            query_list[i] ← n
        end if
    end for
    // Recursively iterate over the remaining list of queries
    return RECONSTRUCT(query_list, data)
end function
```

has a significant impact on the expected cost per claim verification. Therefore, CEDAR uses cost-based optimization to determine an optimal order of verification approaches and an optimal number of retries. We introduce notations and assumptions for this section in Section 6.1 and derive models to assess the expected cost and success probability for verification schedules in Section 6.2. Next, we analyze those models in Section 6.3 to derive formal properties that motivate the scheduling algorithm presented in Section 6.4.

6.1 Notations and Assumptions

We represent verification schedules as a vector v where each vector component corresponds to a verification approach. CEDAR tries verification approaches in the order in which they appear in the verification schedule. We denote the i-th element of such vectors by v_i and ranges of components by $v_{i..j}$. Given a specific verification approach v_i, we denote by $C(v_i)$ the expected costs of verification. Expected costs are due to invocations of the associated language model. We can use profiling to estimate costs, based on the average number of tokens consumed per claim. Also, by $A(v_i)$, we denote the accuracy of the verification approach. This is the probability that the verification approach succeeds. We make several simplifying assumptions to facilitate the analysis.

Assumption 1. *The success probabilities of different retries with the same verification approach are uncorrelated.*

While simplifying, this assumption is reasonable since all verification approaches are based on language models that we use with a certain degree of randomization (i.e., applying the language model twice to the same input can lead to different outcomes).

Assumption 2. *The success probabilities of different verification approaches are uncorrelated.*

This assumption is simplifying as well. However, it is expensive to obtain information on the correlation of success probabilities between different verification approaches via profiling. Assessing the cost of our independence assumptions [11] demonstrates that our models perform well enough to enable effective scheduling, despite these simplifying assumptions.

6.2 Cost and Quality Model

We analyze the expected cost of a verification schedule.

Theorem 6.1. *The expected cost of a verification schedule v, $C(v)$, is given as $C(v) = \sum_i C(v_i) \cdot \prod_{j<i}(1 - A(v_j))$.*

Proof. We can express the expected verification cost via the following formula: $C(v) = \sum_i C(v_i) \cdot \Pr(v_i : \textit{applied})$. Here, $\Pr(v_i : \textit{applied})$ denotes the probability that v_i is applied. As CEDAR tries verification approaches in the order in which they appear in the schedule, approach v_i is only applied if all prior approaches (v_j for $j < i$) fail. The probability that approach v_j fails is given as $1-A(v_j)$. Due to our independence assumptions (Assumptions 1 and 2), the probability that all prior approaches fail is given as $\prod_{j<i}(1-A(v_j))$. Substituting $\Pr(v_i : \textit{applied})$ leads to the postulated formula. □

Next, we analyze the probability of successful verification.

Theorem 6.2. *The accuracy of a verification schedule v, $A(v)$, is given as $A(v) = 1 - \prod_i(1 - A(v_i))$.*

Proof. The probability that verification fails with approach v_i is given as $1 - A(v_i)$. We assume that different tries are statistically independent due to Assumptions 1 and 2. Therefore, the probability that all verification approaches fail is given as $\prod_i(1-A(v_i))$. Finally, the probability that at least one approach succeeds is given as the complement, i.e., $1 - \prod_i(1 - A(v_i))$. □

6.3 Model Analysis

A cost function satisfies the principle of optimality if optimal solutions are composed of optimal solutions to sub-problems. We show that this applies to our scenario.

Theorem 6.3. *Replacing a prefix of a verification schedule with one with at least equal accuracy and at most the same cost cannot worsen any metric for the entire schedule.*

Proof. Let v be the original verification schedule (which corresponds to a sequence of calls to verification methods). Let k be the length of the prefix to replace, c_o and f_o the cost and failure probability of the original prefix, and c_n and f_n the corresponding metrics for the new prefix. It is $c_o \geq c_n$ and $f_o \geq f_n$. According to Theorem 6.2, the accuracy before replacement is $1 - f_o \cdot \prod_{i>k}(1-A(v_i))$ and therefore lower or equivalent to the accuracy after replacement: $1 - f_n \cdot \prod_{i>k}(1 - A(v_i))$ (since $f_n \leq f_o$). According to Theorem 6.1, the cost before replacement can be expressed as $c_o + f_o \cdot \sum_{i>k} C(v_i) \cdot \prod_{k<j<i}(1-A(v_j))$. After replacement, the cost becomes $c_n + f_n \cdot \sum_{i>k} C(v_i) \cdot \prod_{k<j<i}(1 - A(v_j))$. Due to $c_n \leq c_o$ and $f_n \leq f_o$, this cost cannot be higher than before. □

To limit the size of the search space, the scheduling algorithm presented in the next section only considers consecutive retries of the same verification approach. While simplifying, we can prove that this approach yields optimal solutions in certain scenarios.

Theorem 6.4. *Let v be a schedule that divides two invocations of the same verification approach by invocations to a different model. There is a schedule with equivalent or better costs that does not divide those invocations.*

Proof. Let $v_i = A$, $v_{i+1} = B$, $v_j = B$, and $v_{j+1} - A$ with $i < j$. Swapping components v_i and v_{i+1} leads to the following change in the cost estimate. Denoting by f_i the probability that all verification approaches before the i-th approach fail, and by c_A, c_B, f_A, f_B the costs and failure probabilities of verification methods A and B respectively, the change in cost due to the swap can be expressed as follows (using Theorem 6.1): $\Delta_i = f_i \cdot (-c_A - f_A \cdot c_B + c_B + f_B \cdot c_A)$. On the other hand, if swapping the verification approaches at positions k and $k+1$ in the schedule and denoting by f_j the probability that all verification approaches before the j-th one fail, the change in cost due to the swap is $\Delta_j = f_j \cdot (c_A + f_A \cdot c_B - c_B - f_B \cdot c_A)$. As f_i and f_j are probabilities and therefore non-negative, it is not possible that both changes lead to an increase in costs (i.e., either $\Delta_i \leq 0$ or $\Delta_j \leq 0$). Therefore, there is a schedule that brings both invocations to method A closer together and has at most equivalent cost to v. As we can swap repeatedly, due to transitivity, there is at least one cost-equivalent schedule, compared to v, that uses A consecutively. Hence, separating invocations of the same approach by invocations to a second does not yield better solutions. □

6.4 Scheduling Algorithm

Algorithm 10 finds optimal verification schedules, given profiling data (which may be generic or specific to a certain class of data summaries), and a lower bound on the resulting accuracy. The latter parameter enables users to reduce verification costs by allowing for a lower accuracy. Profiling requires labeled data, but according to our experiments discussed in Section 7.3.3, the performance penalty of reusing profiling results across verification domains is limited.

The algorithm uses dynamic programming and shares some similarities with Selinger's classical algorithm for join ordering [29]. However, it differs by its search space as we choose not only a permutation but also the number of tries for each verification method (which includes the possibility of avoiding a certain method altogether). Also, it differs by the presence of multiple metrics, namely

Algorithm 10 Optimal Scheduling Algorithm

```
V is the set of verification methods
m is the maximal number of retries
// Generate an optimal verification schedule,
// given profiling data and an accuracy constraint.
function SCHEDULE(profiling_data, min_A)
    // Initialize entries for single verification methods
    dpTable ← {}
    for v ∈ V do
        dpTable[v] ← ∅
        for k ← 0..m do
            dpTable[v] ← dpTable[v] ∪ {[⟨v, k⟩]}
        end for
    end for
    // Iterate over the number of verification schemes
    for l ← 2..|V| do
        // Iterate over subsets of verification schemes
        for S ⊆ V : |S| = l do
            // Iterate over last verification scheme in schedule
            for v ∈ S do
                // Remove last scheme from set
                S′ ← S \ {v}
                // Iterate over Pareto-optimal solutions
                for p ∈ dpTable[S′] do
                    // Iterate over number of retries
                    for k ∈ 0..m do
                        // Create new verification schedule
                        n ← p ∘ ⟨v, k⟩
                        // Prune with newly generated schedule
                        PRUNE(dpTable[S], n)
                    end for
                end for
            end for
        end for
    end for
    return SELECTSCHEDULE(dpTable[V], min_A)
end function
```

cost and accuracy, which require custom handling during pruning and the final schedule selection step.

First, Algorithm 10 initializes the $dpTable$ variable. This variable maps subsets of verification methods to schedules using those verification methods exclusively (while differing by the associated number of tries). The variable only stores schedules that are Pareto-optimal, considering the two metrics cost and accuracy. Algorithm 10 represents schedules as an (ordered) list of tuples. Each tuple represents the invocation of one specific verification method with a given number of tries (which may also be zero, meaning that the method is, in fact, not used). During initialization, Algorithm 10 maps each verification method to schedules that use the method with different numbers of retries. At this stage, pruning is unnecessary since all such schedules are guaranteed to be Pareto-optimal (assuming a non-zero success probability and cost).

Second, Algorithm 10 constructs schedules that use more and more different verification methods. In the outermost loop, it iterates over the number of verification methods that a schedule can use (ending with the cardinality of V, i.e., the total number of verification methods available). For each number of methods, the algorithm iterates over subsets (variable S) of verification methods with the corresponding cardinality. For each of those subsets, it iterates over the verification method that is used last. For that method, the algorithm iterates over the number of tries, starting with zero and ending with the maximal number of tries (m). Finally, Algorithm 10 iterates over all Pareto-optimal schedules, stored for the remaining subset S' of verification methods (excluding the verification method, selected as the last method). For each Pareto-optimal schedule, it appends the tuple representing the invocation of the last verification method, using the given number of tries. This yields a new schedule that uses all of the verification methods in S.

The algorithm prunes the content of $dpTable[S]$ with the newly generated schedule. This means the new schedule is inserted and all schedules that are not Pareto-optimal (including, possibly, the newly generated schedule) are discarded. The PRUNE function (whose pseudo-code is omitted) is specialized to our scenario which involves two cost metrics. It first sorts all schedules by their accuracy, then scans sorted schedules to efficiently discard schedules that do not balance out reduced accuracy with lower costs. After iterations are complete, $dpTable[V]$ contains Pareto-optimal schedules.

THEOREM 6.5. *Algorithm 10 generates optimal schedules.*

PROOF. Algorithm 10 considers all permutations of verification methods and, for each method, all possible numbers of tries. It only discards verification schedules that are Pareto-dominated. However, due to Theorem 6.3, replacing the prefix of a verification schedule with one that has better or equivalent cost and accuracy cannot worsen any of those metrics for the entire schedule either. Therefore, the algorithm generates Pareto-optimal schedules. □

The final schedule is selected from the Pareto-optimal schedules in $dpTable[V]$ (Function SELECTSCHEDULE, pseudo-code omitted), using the following rules. First, if available, we restrict our focus to schedules that match the user-specified accuracy constraint. Otherwise, only schedules with maximal accuracy are considered. Second, to achieve the highest possible accuracy, we perform additional filtering to account for the simplifying assumptions in our cost model. According to Assumption 2, our accuracy model treats tries by different verification methods as statistically independent. However, in practice, using more diverse verification methods increases success chances, compared to many retries with the same method. Therefore, we filter the remaining schedules to the ones that use the maximal number of different verification methods. Among those, we ultimately select the one with minimal estimated costs.

7 EXPERIMENTAL EVALUATION

Section 7.1 describes the experimental setup, Section 7.2 compares CEDAR to baselines, and Section 7.3 studies scenario variants.

7.1 Experimental Setup

We evaluate all baselines on three diverse collections of data sets. The AggChecker data set introduced in prior work [14] contains 56 data summaries containing 392 numerical claims in total, and covers publicly available articles from several newspapers (including 538 and the New York Times), as well as summaries of Stack Overflow developer surveys and Wikipedia articles. The TabFact data set [34] is a large-scale data set featuring 118k natural language statements labeled as either 'entailed' or 'refuted' based on 16k relational tables from Wikipedia. We use a sample of 100 numerical claims based on 28 tables derived from TabFact. The WikiText data set, which evaluates the performance of the system on textual claims, features 50 claims extracted from 14 Wikipedia articles. While all other data sets use single-table databases for verification, we introduce JoinBench, a benchmark where claim queries require joins. JoinBench was constructed by decomposing three of the original single-table schemas from AggChecker into a total of 23 tables via schema normalization (while reusing the associated AggChecker claims). Claims in AggChecker and WikiText are taken from real-world documents (using data sets referenced by the authors themselves). TabFact uses Wikipedia tables and human-annotated claims. All data sets are publicly available on GitHub [12].

We compare CEDAR to multiple prior systems for data-driven fact-checking. These include AggChecker [14] and TAPEX [22]. AggChecker verifies claims by translating them to SQL queries whereas TAPEX applies language models to both, tabular data and text claims. Additionally, we leverage text-to-SQL translation methods, using the "Create Table + Select 3" (P1) prompt template (which is described in detail in prior work [26] and performed best among multiple templates) and the text-to-SQL translation prompt template (P2) proposed by OpenAI [4]. For both templates, we first translate the claim into a question and then apply GPT-3.5 to translate that question to SQL, using both aforementioned prompt templates. We use the same method as CEDAR to assess whether the translated SQL query supports or refutes the claim.

As in prior work [14], we measure the quality of verification results according to three metrics: recall (the ratio of incorrect claims identified), precision (the ratio of claims marked as incorrect that are indeed incorrect), and the F1 Score (combining precision and recall). We also report execution time and monetary execution fees (incurred for LLM invocations, the dominant cost component).

A demonstration of CEDAR [13] was presented at SIGMOD 2025. All experiments are executed on a server with Intel Core i7-1355U CPU with 16 GB of main memory. CEDAR is implemented in Python, using the LangChain framework. CEDAR uses DuckDB to execute SQL queries. Unless noted otherwise, we use CEDAR with a high accuracy threshold of 99%. CEDAR uses the following verification approaches: the one-shot approach from Section 5.2 with GPT-3.5 and GPT-4o, and the agent approach from Section 5.3 with GPT-4o and GPT-4.0. For each approach, we use a temperature of zero for the initial invocation and a temperature of 0.25 (for one-shot approaches) or 0.5 (for agent approaches) when invoking the approach a second time on the same claim. The extended technical report [11] contains additional experiments and technical details.

7.2 Comparison to Baselines

Table 2 presents a comparative analysis of model performance on the AggChecker, TabFact and WikiText data sets using precision, recall, and F1 score metrics. CEDAR consistently outperforms all

Table 2: Comparing result quality of CEDAR and baselines.

Dataset	Baseline	CEDAR	AggC	TAPEX	P1	P2
Agg Checker	**Precision**	**59.7**	36.2	0	15	15
	Recall	**89.6**	70.8	0	64	70
	F1 score	**71.7**	47.9	0	24	24
Tab Fact	**Precision**	87.9	50	**88.5**	45.4	41.9
	Recall	85.3	34.6	71.9	88.2	**91.2**
	F1 score	**86.6**	40.9	79.3	60	57.4
Wiki Text	**Precision**	33.3	-	**100**	0	4.5
	Recall	**100**	-	18	0	**100**
	F1 score	**50**	-	30.5	0	28.64

other approaches, achieving the highest F1 scores across all data sets. On the AggChecker data set, CEDAR improves the F1 score from 48% (AggChecker, the runner-up) to 72%, while on the TabFact data set, it enhances the F1 score from 79% (TAPEX, the runner-up) to 87%. On the WikiText data set containing textual claims, CEDAR improves the F1 score from 30.5% (TAPEX, the runner-up) to 50%. The AggChecker baseline does not support textual claims. An interesting observation is that TAPEX, which performs as the second-best model on the TabFact data set, has the worst performance on the AggChecker data set. This discrepancy arises because TAPEX uses a table-flattening technique that is effective for smaller tables but becomes infeasible as table sizes increase. Text-to-SQL translation methods are not designed to exploit claim values for guidance regarding the correctness of query translation (different from the fact-checking baselines), resulting in significantly lower F1 scores. With the high accuracy threshold of 99%, CEDAR incurs costs of $18.12 to verify all 392 claims of the AggChecker data set, $1.46 for TabFact and $1.9 for WikiText. This is significantly higher than for the other baselines exploiting LLMs as a service, namely P1 and P2. As shown at the end of this section, costs can be tuned by lowering the accuracy threshold.

Figure 5 compares cost–quality and throughput–quality tradeoffs of baselines to single-stage verification and CEDAR's multi-stage verification approach using different accuracy thresholds, evaluated on the AggChecker [14] data set. CEDAR spans the full Pareto frontier in the Cost-F1 score space, and realizes Pareto-optimal trade-offs in the Throughput–F1 score space, as the optimization focuses on cost rather than time. CEDAR achieves significant improvements in cost and F1 score, compared to GPT-4.0 agent (the baseline with highest F1 score).

7.3 Further Analysis

7.3.1 Unit Conversions. In some cases, units used in claims differ from the ones used in source data (e.g., to increase clarity for specific reader groups). If so, verifying claims requires unit conversions. We evaluated our system's ability to handle such conversions using a benchmark [12] of 20 claims drawn from eight Wikipedia articles. The system achieves an F1 score of 94.7% when the claim units align with those in the dataset, and maintains a strong performance of 88.9% even when unit conversions are required. In Figure 6, each bar represents the difference in F1 score when changing the data

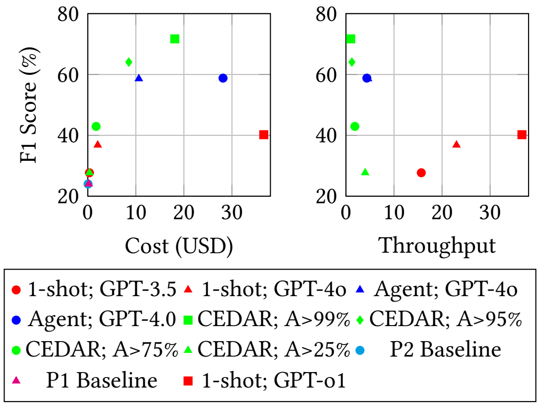

Figure 5: Comparing cost-quality tradeoffs of single-stage and multi-stage verification on AggChecker [14] data set.

Table 3: Query complexity statistics across data sets.

Data set	Joins	GroupBy	SubQ	Agg	Cols
AggChecker	0/ 0	0.01/ 1	0.54/ 2	0.99/ 12	1.3/ 2
TabFact	0/ 0	0/ 0	0.09/ 2	0.63/ 1	1.05/ 2
WikiText	0/ 0	0.22/ 1	0.33/ 3	0.51/ 3	1.33/ 4
JoinBench	0.62/ 3	0/ 0	0.52/ 2	0.76/ 2	1.5/ 2

such that unit conversions are required for verification. For most documents, mismatched units have minimal impact on verification accuracy (except for one document containing a single claim that CEDAR is unable to verify when unit conversions are required).

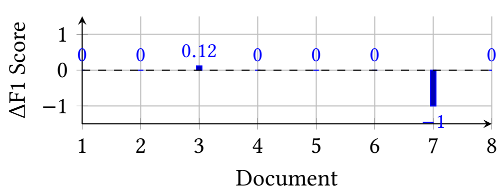

Figure 6: Change in F1 score (ΔF1) due to unit conversions.

7.3.2 Query Complexity. Table 3 reports detailed statistics (per-query average/maximum) on the complexity of SQL queries across benchmarks. JoinBench re-uses a subset of claims from AggChecker while normalizing the corresponding database such that joins are required. Normalization does not change the F1 score achieved by CEDAR (100% in both cases) but increases costs from $1.2 to $3.7, showing that verification using join queries often requires more expensive verification methods.

7.3.3 Effect of Distribution Shifts on Profiling. We evaluate the robustness of our approach under distribution shifts. We calculate optimal schedules for each single document in the AggChecker data set, resulting in eight distinct verification schedules. Then, we apply each schedule to subsets of the AggChecker claims associated with different domains (defined by the claim source: 538, StackOverflow, NYTimes, and Wikipedia). Figure 7 shows the change in F1 score and cost overheads when applying schedules across domains. While domain-specific profiling and optimization yield optimal results, the performance penalty of generalization is limited (cost overheads of less than factor 2 and F1 loss of less than 0.1 in 80% of cases). However, profiling may have to be redone as models evolve.

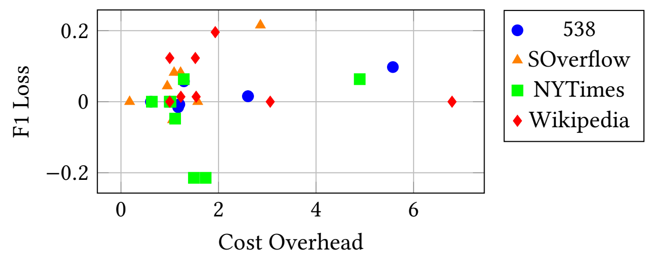

Figure 7: Cost Overhead vs. F1 Loss across profiling results.

8 RELATED WORK

We address the problem of data-driven fact-checking from relational data. We compare with several prior methods [9, 14, 22] for the same problem in our experiments. Other work addresses the same problem but under different assumptions. E.g., Scrutinizer [15, 16] assumes that domain-specific classifiers for claim-to-SQL translation are trained via fact-checkers. TabFact [34] and TFV [1] also assume that labeled training data for translation is available. CEDAR does not require user-provided training data for claim-to-SQL translation. Manual labels are required for profiling. Our work is complementary to prior methods [6] targeting related problems such as identifying check-worthy claims [7, 8, 18], studying the robustness of data-related claims via query perturbations [32, 35, 36], or automated fact-checking methods that use non-relational data sources for verification, including knowledge bases [10, 30], or unstructured data [5, 20, 25, 31]. As CEDAR focuses on automated fact-checking, it is complementary to methods that aim at facilitating fact-checking for human fact-checkers [23, 24]. Our work relates to methods for text-to-SQL translation [19, 21, 27, 28]. However, the problem of claim-to-SQL translation differs, e.g., by the presence of a claimed query result. The multi-stage verification approach applied by CEDAR is ultimately enabled by claimed results, providing the system with a strong signal as to whether or not translation in prior stages was successful.

9 CONCLUSION

We introduced CEDAR, a system for cost-efficient data-driven claim verification. Compared to prior work, CEDAR is more accurate and reduces costs via multi-stage execution and cost-based scheduling.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Mingke Chai, Zihui Gu, Xiaoman Zhao, and Ju Fan. 2021. TFV : A Framework for Table-Based Fact Verification. *Data Engineering Bulletin* 59 (2021), 39–51.

[2] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain Last accessed July 11, 2025.

[3] FiveThirtyEight. 2014. Should Travelers Avoid Flying Airlines That Have Had Crashes in the Past? https://fivethirtyeight.com/features/should-travelers-avoid-flying-airlines-that-have-had-crashes-in-the-past/ Last accessed July 11, 2025.

[4] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *CoRR* abs/2308.15363 (2023).

[5] Jiahui Geng, Yova Kementchedjhieva, Preslav Nakov, and Iryna Gurevych. 2024. Multimodal Large Language Models to Support Real-World Fact-Checking. arXiv:2403.03627 [cs.CL] https://arxiv.org/abs/2403.03627

[6] Zhijiang Guo, Michael Schlichtkrull, and Andreas Vlachos. 2022. A Survey on Automated Fact-Checking. *Transactions of the Association for Computational Linguistics* 10 (2022), 178–206. https://doi.org/10.1162/tacl_a_00454

[7] Naeemul Hassan, Fatma Arslan, Chengkai Li, and Mark Tremayne. 2017. Toward automated fact-checking: detecting check-worthy factual claims by ClaimBuster. In *SIGKDD*. 1803–1812.

[8] Naeemul Hassan, Chengkai Li, and Mark Tremayne. 2015. Detecting check-worthy factual claims in presidential debates. In *CIKM*. 1835–1838. https://doi.org/10.1145/2806416.2806652

[9] Naeemul Hassan, Anil Nayak, Vikas Sable, Chengkai Li, Mark Tremayne, Gensheng Zhang, Fatma Arslan, Josue Caraballo, Damian Jimenez, Siddhant Gawsane, Shohedul Hasan, Minumol Joseph, and Aaditya Kulkarni. 2017. ClaimBuster: the first-ever end-to-end fact-checking system. *Proceedings of the VLDB Endowment* 10 (08 2017), 1945–1948. https://doi.org/10.14778/3137765.3137815

[10] Viet-Phi Huynh and Paolo Papotti. 2019. Buckle: evaluating fact checking algorithms built on knowledge bases. *VLDB* 12, 12 (2019), 1798–1801.

[11] Tharushi Jayasekara and Immanuel Trummer. 2025. CEDAR: A System for Cost-Efficient Data-Driven Claim Verification. https://github.com/TharushiJay/CEDAR/blob/main/Extended%20Technical%20Report.pdf Available online, last accessed July 11, 2025.

[12] Tharushi Jayasekara and Immanuel Trummer. 2025. CEDAR: Code Repository. https://github.com/TharushiJay/CEDAR Available online, last accessed July 11, 2025.

[13] Tharushi Jayasekara and Immanuel Trummer. 2025. Demonstrating CEDAR: A System for Cost-Efficient Data-Driven Claim Verification. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 135–138. https://doi.org/10.1145/3722212.3725098

[14] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Verifying text summaries of relational data sets. In *Proceedings of the 2019 International Conference on Management of Data*. 299–316.

[15] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: a mixed-initiative approach to large-scale, data-driven claim verification. *Proc. VLDB Endow.* 13, 12 (July 2020), 2508–2521. https://doi.org/10.14778/3407790.3407841

[16] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: Fact Checking Statistical Claims. *PVLDB* 13, 12 (2020), 2965–2968. https://doi.org/10.14778/3415478.3415520

[17] Georgios Karagiannis, Immanuel Trummer, Saehan Jo, Shubham Khandelwal, Xuezhi Wang, and Cong Yu. 2020. Mining an "Anti-Knowledge Base" from Wikipedia Updates with Applications to Fact Checking and Beyond. *PVLDB* 13, 4 (2020), 561–573. https://doi.org/10.14778/3372716.3372727

[18] Lev Konstantinovskiy, Oliver Price, Mevan Babakar, and Arkaitz Zubiaga. 2021. Toward Automated Factchecking: Developing an Annotation Schema and Benchmark for Consistent Automated Claim Detection. *Digital Threats* 2, 2, Article 14 (April 2021), 16 pages. https://doi.org/10.1145/3412869

[19] Fei Li and HV Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*. 709–712.

[20] Miaoran Li, Baolin Peng, Michel Galley, Jianfeng Gao, and Zhu Zhang. 2024. Self-Checker: Plug-and-Play Modules for Fact-Checking with Large Language Models. In *Findings of the Association for Computational Linguistics: NAACL 2024 - Findings*. 163–181. https://doi.org/10.18653/v1/2024.findings-naacl.12 arXiv:2305.14623

[21] Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. 2023. A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability. arXiv:2303.13547 [cs.CL] https://arxiv.org/abs/2303.13547

[22] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table Pre-training via Learning a Neural SQL Executor. In *International Conference on Learning Representations*. https://openreview.net/forum?id=O50443AsCP

[23] Preslav Nakov, David Corney, Maram Hasanain, Firoj Alam, Tamer Elsayed, Alberto Barrón-Cedeño, Paolo Papotti, Shaden Shaar, and Giovanni Da San Martino. 2021. Automated fact-checking for assisting human fact-checkers. *arXiv preprint arXiv:2103.07769* (2021).

[24] Thanh Tam Nguyen, Matthias Weidlich, Hongzhi Yin, Bolong Zheng, Quoc Viet Hung Nguyen, and Bela Stantic. 2018. User guidance for efficient fact checking. *Proceedings of the VLDB Endowment* 12, 8 (2018), 850–863. https://doi.org/10.14778/3324301.3324303

[25] Dorian Quelle and Alexandre Bovet. 2024. The perils and promises of fact-checking with large language models. *Frontiers in Artificial Intelligence* 7 (2024). https://doi.org/10.3389/frai.2024.1341697

[26] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498* (2022).

[27] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Ozcan. 2016. ATHENA: An ontology-driven system for natural language querying over relational data stores. *VLDB* 9, 12 (2016), 1209–1220.

[28] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *EMNLP*. 9895–9901. https://doi.org/10.18653/v1/2021.emnlp-main.779 arXiv:2109.05093

[29] PG G Selinger, MM M Astrahan, D D Chamberlin, R A Lorie, and T G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34. http://dl.acm.org/citation.cfm?id=582095.582099

[30] Baoxu Shi and Tim Weninger. 2016. Fact Checking in Heterogeneous Information Networks. In *WWW*. 101–102. https://doi.org/10.1145/2872518.2889354

[31] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. FEVER: a Large-scale Dataset for Fact Extraction and VERification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, Marilyn Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, New Orleans, Louisiana, 809–819. https://doi.org/10.18653/v1/N18-1074

[32] Brett Walenz and Jun Yang. 2016. Perturbation analysis of database queries. *VLDB* 9, 14 (2016), 1635–1646.

[33] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in neural information processing systems* 33 (2020), 5776–5788.

[34] Jianshu Chen Yunkai Zhang Hong Wang Shiyang Li Xiyou Zhou Wenhu Chen, Hongmin Wang and William Yang Wang. 2020. TabFact : A Large-scale Dataset for Table-based Fact Verification. In *International Conference on Learning Representations (ICLR)*. Addis Ababa, Ethiopia.

[35] You Wu, Pk Agarwal, C Li, J Yang, and C Yu. 2014. Toward computational fact-checking. *VLDB* 7, 7 (2014), 589–600. https://doi.org/10.14778/2732286.2732295

[36] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2017. Computational fact checking through query perturbations. *Trans. on Database Systems* 42, 1 (2017), 1–41. https://doi.org/10.1145/2996453

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]