QURE: AI-Assisted and Automatically Verified UDF Inlining (Extended Version)

TARIQUE SIDDIQUI, Microsoft Research, USA
ARND CHRISTIAN KÖNIG, Microsoft Research, USA
JIASHEN CAO, Georgia Tech, USA*
CONG YAN, Snowflake, USA*
SHUVENDU K. LAHIRI, Microsoft Research, USA

User-defined functions (UDFs) extend the capabilities of SQL by improving code reusability and encapsulating complex logic, but can hinder the performance due to optimization and execution inefficiencies. Prior approaches attempt to address this by rewriting UDFs into native SQL, which is then inlined into the SQL queries that invoke them. However, these approaches are either limited to simple pattern matching or require the synthesis of complex verification conditions from procedural code, a process that is brittle and difficult to automate. This limits coverage and makes the translation approaches less extensible to previously unseen procedural constructs. In this work, we present QURE, a framework that (1) leverages large language models (LLMs) to translate UDFs to native SQL, and (2) introduces a novel formal verification method to establish equivalence between the UDF and its translation. QURE uses the semantics of SQL operators to automate the derivation of verification conditions, in turn resulting in broad coverage and high extensibility. We model a large set of imperative constructs, particularly those common in Python and Pandas UDFs, in an intermediate verification language, allowing for the verification of their SQL translation. In our empirical evaluation of Python and Pandas UDFs, equivalence is successfully verified for 88% of UDF-SQL pairs (the rest lack semantically-equivalent SQLs) and LLMs correctly translate 84% of the UDFs. Executing the translated UDFs achieves median performance improvements of 23× on single-node clusters and 12× on 12-node clusters compared to the original UDFs, while also significantly reducing out-of-memory errors.

CCS Concepts: • **Information systems** → **Query operators**; **Query optimization**.

Additional Key Words and Phrases: UDF, query rewriting, query optimization, formal verification, LLMs.

ACM Reference Format:
Tarique Siddiqui, Arnd Christian König, Jiashen Cao, Cong Yan, and Shuvendu K. Lahiri. 2025. QURE: AI-Assisted and Automatically Verified UDF Inlining (Extended Version). 3, 1 (SIGMOD), Article 66 (February 2025), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Database systems (e.g., PostgreSQL, SQL Server, Spark) offer procedural extensions such as user-defined functions (UDFs) with support for various programming languages (e.g., Python, T-SQL, C#) to broaden the scope of declarative SQL. These extensions improve modularity, reusability, and

* Work done while at Microsoft.
Authors' Contact Information: Tarique Siddiqui, Microsoft Research, USA, tasidd@microsoft.com; Arnd Christian König, Microsoft Research, USA, chrisko@microsoft.com; Jiashen Cao, Georgia Tech, USA *, jiashenc@gatech.edu; Cong Yan, Snowflake, USA *, congyan.me@gmail.com; Shuvendu K. Lahiri, Microsoft Research, USA, shuvendu@microsoft.com.

ACM XXXX-XXXX/2025/2-ART66
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1. A while loop and its loop invariants. Analyzing the procedural code to *automatically synthesize* these invariants is challenging and error-prone.

// An example while loop doing filtering (extracted from a real UDF)

```
output_index := 0, loop_index := 0;
while (loop_index < key_list_len) {
  c := key_list[loop_index];
  if (c == 'A'){
    filteredlist[output_index] := c;
    output_index := output_index + 1;
  }
  loop_index := (loop_index + 1);
}
```

// Loop invariants for the above loop

(1) Valid range of loop_index: $0 \leq$ loop_index $\leq$ key_list_len.
(2) Valid range of output_index: $0 \leq$ output_index $\leq$ loop_index.
(3) Every element in filteredlist up to output_index is "A" and exists in key_list before loop_index:

$$\forall l \in \mathbb{Z}, (0 \leq l < \text{output\_index}) \implies (\text{filteredlist}[l] = \text{"A"} \wedge \exists k \in \mathbb{Z}, (0 \leq k < \text{loop\_index}) \wedge (\text{key\_list}[k] = \text{filteredlist}[l]))$$

(4) If an element of the key_list is "A" and is located before loop_index, then it must also appear in the filteredlist.

$$\forall k \in \mathbb{Z}, ((0 \leq k < \text{loop\_index}) \wedge (\text{key\_list}[k] = \text{"A"})) \implies (\exists l \in \mathbb{Z}, (0 \leq l < \text{output\_index}) \wedge (\text{key\_list}[k] = \text{filteredlist}[l]))$$

(5) The count of "A" elements in key_list up to index loop_index must equal the size of filteredlist:
output_index $= \sum_{k=0}^{\text{loop\_index}-1} [\text{key\_list}[k] = \text{"A"}]$ where $[\text{cond}]$ is 1 if the cond is true, and 0 otherwise.

provide easier specification of complex logic. However, UDFs are known to have performance limitations. For one, it is challenging to infer UDF properties, such as output cardinality and execution cost during query optimization [26]. UDFs can be also slow to execute due to computational and data transfer overheads, particularly when UDF is written in a language different from the SQL dialect supported by the database.

Limitations of the state of the art. To address these issues, considerable research has been done on the translation of UDF into SQL. A large body of work, e.g., [24, 25, 31, 33–35, 50, 51], aims to directly translate procedural constructs to SQL expressions via pattern matching and rule-based transformations. Such techniques have low overhead, but due to the complex syntax and semantics of procedural constructs (e.g., see the `while` loop in Table 1), these techniques may struggle to translate them to corresponding built-in SQL operations or functions; e.g., *Froid* [51] limits the translation to a set of simple filter and projection expressions. Some techniques [25, 31, 35] can translate iterative constructs into SQL expressions using recursive CTEs or customized aggregates; however, avoiding the use of these constructs (where possible) typically results in better performance (which we demonstrate experimentally in Section 7.3).

In addition, programming language techniques [17, 61, 62] with relatively broader coverage have been developed that first *summarize* the semantics of imperative code via *Hoare triples* [36] and then check their correctness, resulting in broader coverage of language constructs. Expressions in Hoare triples are easier to then translate to equivalent SQL than lower-level procedural constructs. As we discuss in more detail in Section 2.2, a Hoare triple $\{A\}\ P\ \{B\}$ asserts that a program P starting with a state satisfying predicate A (the *pre-condition*) results in a state satisfying predicate B (the *post-condition*) on termination. If P contains loops, the above techniques also synthesize *loop invariants*, which are conditions that must hold for any iteration of the loop(s). The correctness of these *verification conditions* (e.g., post-conditions, invariants) in Hoare-triples is *verified* using formal verification tools that leverage SMT solvers [10] such as Z3 [14]. Once verified, post-conditions are translated via rules to a SQL query that is equivalent to the original code.

However, the automatic synthesis of verification conditions *from procedural code* for even seemingly straightforward operations (e.g., row-wise transformations, filters, and aggregation) necessitates complex analysis that can be error prone, fragile, and hard to automate [15, 16, 27]. For

instance, Table 1 depicts the code for a `while` loop used in a real UDF that corresponds to a filter operation in SQL, along with a list of necessary invariants. Inferring these invariants, especially in (3) and (4), from the loop's code itself can be challenging.

Another challenge in both rule-based and programming language-based techniques is modeling rich semantic languages like Python (which is the focus of this work). Python UDFs often involve diverse procedural constructs such as lists, dictionaries, various loops, DataFrames, and external library function calls whose source code may be unavailable or highly complex. Additionally, it is nearly impossible to anticipate all constructs (in both UDF and SQL) in advance, so the framework must be easily *extensible* to accommodate new constructs without significant rewrites.

End to end AI-assisted framework for UDF inlining. In this work, we develop *QURE* ("*QUery REwriting*"), a LLM-driven framework that performs automatic inlining of translated UDFs within SQL queries. Our focus is on translating *Python* and *Pandas* [6] UDFs in Apache Spark into equivalent *SparkSQL* expressions. Before we describe how QURE addresses the above challenges, we give a brief overview of its end-to-end functionality.

QURE takes a UDF and a SQL query (the "*parent query*") using the UDF as inputs. Initially, an LLM, such as GPT-4o, translates the UDF into SQL, referred to as UDF SQL. We have developed few-shot prompting techniques tailored to different categories of UDFs (discussed in Section 2.1) and use context from the parent query and database schema for translation. After translation, the system verifies the equivalence of the UDF and the SQL translation, using a novel verification approach that uses Hoare-triples that leverages the semantics of SQL to address the challenges of synthesizing verification conditions as we discuss shortly. Upon successful verification, QURE inlines the UDF SQL with the parent query, creating a final unified SQL query for execution. If the verification fails, QURE preserves the original UDF, thereby ensuring the correctness of the query's results. We now discuss how we address the aforementioned challenges.

LLMs allow for broader coverage of UDF to SQL Translation. Trained on a large corpus of procedural and declarative code, LLMs excel in code comprehension, synthesis, and translation [45, 46, 54]. Our evaluation (Section 7) shows that LLMs can translate a diverse range of procedural constructs (e.g., lists, dictionaries, loops, and DataFrames) into equivalent SQL expressions compared to prior pattern matching-based techniques, e.g., [51]. Developing a comprehensive set of rules to cover all procedural code patterns for SQL expressions is possible but challenging, as these rules can be brittle, hard to update, and difficult to compose for complex SQL expressions. In contrast, LLMs continue to improve (e.g., from GPT-3.5 to GPT-4o) for such translation tasks. However, the challenge lies in verifying the equivalence between the UDF and the inferred SQL.

Leveraging SQL to automate synthesis of verification conditions. The primary contribution of our work lies in automatically synthesizing the verification conditions required for equivalence verification from the logical query plan of the inferred SQL query. Our observation is that, given a candidate SQL query (which is inferred by LLM in QURE but can be obtained through alternative sources as well), we can analyze the SQL (instead of the procedural code in the UDF) to infer verification conditions. Specifically, we compile the SQL query into a *logical plan* [3] which is represented as an acyclic graph of (logical) operators (and functions). Since SQL operations in databases have well-defined semantics, the verification conditions (capturing the relationships between inputs and ouputs) of SQL operators can be precisely *modeled in advance* (offline), and can be *functionally composed* similar to operators in a logical plan to synthesize post-conditions and invariants (discussed in detail in Section 4). This process is simpler and less error-prone, requiring lighter-weight syntactic analysis of the procedural code. To the best of our knowledge, this is the first approach that addresses the challenges of automatically verifying equivalence between (LLM-generated) SQLs and UDFs, using a verification approach that leverages the semantics of SQL.

Modeling of imperative code constructs. Verifying Hoare-triples necessitates representing procedural code in UDFs in a form amenable to reasoning for verification. We do this by modeling a wide range of common procedural constructs, such as lists, dictionaries, loops, and DataFrames, in the intermediate verification language *Boogie* [55]. Boogie's backend translates this code into logical formulas for the SMT solver. We also leverage *uninterpreted functions* [12] (see in Section 2.3) to abstract the complexity of procedural constructs, such as library functions, characterizing only the behavior necessary for verification. Additionally, QURE exposes an expressive *function repository* where frequently occurring functions in UDFs and their SQL equivalents (along with necessary semantics) can be easily registered as uninterpreted functions to facilitate verification.

Extensibility. QURE is extensible in two ways. First, the verification conditions only need to be specified at the level of SQL operators or functions as discussed above. New operators or functions can be supported by adding their corresponding (parameterized) verification conditions without requiring to alter existing operations or the verification approach. Second, the function repository provides a concise representation to add new functions without affecting the rest of QURE.

Summary of Results. For empirical evaluation, we consider a diverse set of UDFs, targeting both generic Python as well as variations of Pandas UDFs. We observe that about 84.8% of these UDFs are correctly translated by LLMs (specifically, GPT-4o). Within this group, QURE verifies the equivalence of UDFs and their SQL translations in 99% of cases and identifies inaccuracies in LLM translations with 100% accuracy. When LLM errors are fixed, QURE covers about 88% of the UDFs, and the rest of the UDFs have no equivalent SQLs. While it is difficult to directly compare with prior approaches (since they target different systems with different UDF characteristics), our comparison with related work (see Section 8) shows that (1) QURE provides a wider coverage of procedural constructs common in Python and Pandas UDFs (see Section 5), (2) can cover more commonly occurring loop-structures for SQL operations including filters, aggregations, joins and their combinations, and (3) can be easily extended to previously unseen functions or operators.

Additionally, UDF inling results in significant performance improvements, including a median improvement of close to 23.7× over single-node configurations and 12.5× in 12-node cluster setups, with each node equipped with 64 GB RAM and 4 CPU cores. Our approach also significantly reduces the frequency of out-of-memory errors often seen for Spark queries containing UDFs. The translation overheads due to LLM inference and formal verification are between 4 to 6 seconds, and for our benchmark queries, our results show that QURE is useful for long-running analytic queries in big data systems such as Spark, for which runtime improvements from translations can be on the order of minutes. QURE is also beneficial in scenarios where queries involving UDFs are tuned offline, e.g., via a co-pilot, or for recurring queries.

2 Background

We first characterize different types of UDFs supported by Apache Spark and then provide a background on formal verification techniques used in QURE.

2.1 Characterizing UDFs in Spark

Our focus is on four common types of UDFs: *Python Scalar*, *Pandas Scalar*, *Pandas Aggregation* and *Pandas Map* UDFs, which we briefly illustrate in the following. All examples shown are real UDFs, collected using the methodology described in Section 7.

(1) Python Scalar UDFs. Python Scalar UDFs are often used for row-wise transformations within SQL queries, maintaining a one-to-one relationship between input and output. Translation of Scalar UDFs can be challenging due to their use of features not directly supported by SQL (e.g., string manipulation, dictionary look-ups, function calls to Python libraries, etc.). Example:

```
SELECT speed(l_shipmode) FROM lineitem
def speed(key):
    key_list = key.split(" ")
    if key_list[0] == "REG":
        return "regular"
    elif key_list[0] in ["AIR", "MAIL"]:
        return "fast"
    else:
        return "slow"
```

(2) Pandas Scalar UDFs. Pandas Scalar UDFs process entire columns as inputs, enabling more efficient analysis. Example:

```
SELECT normalize(l_quantity) FROM lineitem
def normalize(dfs):
    return (dfs - dfs.mean()) / dfs.std()
```

(3) Pandas Aggregation UDFs. Pandas Aggregation UDFs are specialized for complex data summarization tasks, accepting multiple inputs to produce a singular, aggregated output. Use-cases include group-wise computations. Example:

```
SELECT range_diff(l_quantity) FROM lineitem
GROUP BY year(l_shipdate)
def range_diff(dfs):
    return dfs.max() - dfs.min()
```

(4) Pandas Map UDFs. Pandas Map UDFs take an entire table as input and output a table. Here, the required table manipulations are typically realized using Pandas DataFrames. Example:

```
SELECT minus_10(l_quantity) FROM lineitem
GROUP BY year(l_shipdate)
def minus_10(df):
    l_q_col = df["l_quantity"]
    return df.assign(l_q=l_q_col - 10)
```

Summary of observed UDF constructs: Abstracting from examples of the four types of Spark UDFs we collected (see Section 7), we found them to commonly include row-wise data transformations, aggregations, string manipulation, regular expressions, dictionaries, and logical conditions. Statistical analyses and mathematical computations are also common. Pandas UDFs frequently involve DataFrame operations as well as Pandas [6] and NumPy [4] function calls. Complex or nested loops are less frequent, but one or more non-nested loops were common, usually for row-wise transformations, filtering, and aggregation. Note that, in contrast to systems like Microsoft SQL Server and PostgreSQL, which allow the integration of SQL with procedural code within the UDF, Spark UDFs are predominantly procedural, and instead use DataFrame [13] functions for various SQL operations. Translation of UDFs containing SQL statements is outside the scope of this work.

2.2 Hoare-Style Verification

Hoare logic is a formal method for reasoning about program correctness using pre-conditions, post-conditions, and invariants. Consider a simple program to calculate the sum of the first n natural numbers:

```
def sum_natural_numbers(n):
    sum = 0
    for i in range(1, n+1):
        sum += i
    return sum
```

- **Pre-condition:** The state that must be true *before* a program executes, e.g., $n \geq 0$.
- **Post-condition:** The state that must be true *after* a program executes, assuming the pre-condition was met, e.g., the return value is the sum of the first n natural numbers, sum == $\frac{n(n+1)}{2}$

- **Loop Invariant:** A condition that remains true a given program point such as the loop head, e.g., at the start of each loop iteration i, sum == $\frac{i(i-1)}{2}$.

The core of Hoare Logic is the Hoare Triple: $\{A\}\ P\ \{B\}$ where A is the pre-condition, P is the program code, and B is the post-condition. It asserts that if A holds before P executes, and P terminates, then B will hold afterwards. For the above program, we formulate the following Hoare Triple for the loop in lines 3-4: {Loop Invariant: sum == $\frac{i(i-1)}{2} \wedge 1 \leq i \leq n$} `sum += i` {sum == $\frac{i(i+1)}{2}$} and for the program: $\{n \geq 0\}$ `sum_natural_numbers`(n) {sum == $\frac{n(n+1)}{2}$}.

Tools for Verification. Several tools [37, 43, 55] provide frameworks for establishing proofs of correctness in Hoare logic. In this work, we translate Python programs into Boogie [55], and express pre-conditions, post-conditions, and invariants using the constructs provided by Boogie's language. These are then checked using a *SMT solver* [10] supported by Boogie. To the best of our knowledge, previous studies have not explored the translation of Python and SQL code into Boogie. Translating Python, in particular, presents several challenges, which we will discuss in Section 5. Additionally, synthesizing post-conditions and invariants is a significant challenge, as Boogie does not automatically infer these elements.

Query-by-Synthesis and Challenges. Prior work, such as QBS [17], synthesizes post-conditions and invariants from procedural code, and once verified, translates post-conditions into equivalent SQL expressions via rules. However, automatically inferring invariants and post-conditions, even for simple loops, is hard, as illustrated in Table 1, making full automation challenging. As we discuss shortly, we address this challenge by leveraging SQL inferred by LLMs and analyzing the SQL-derived plan, simplifying the construction of post-conditions and invariants. The semantics of SQL operators can be precisely modeled in advance (offline) and incorporated into procedural code through lightweight analysis (e.g., input and output variables in the code segment representing the operator), bypassing the need for extracting complex logical assertions from the procedural code.

2.3 Uninterpreted Functions

In formal verification and theorem proving, uninterpreted functions are used to model functions whose complete implementation details are unknown or irrelevant. Specific properties necessary for verification can be characterized for these functions via axioms, which the verifier can assume without detailed implementation knowledge. For example, if we need to verify properties such as commutativity or idempotence for a function $f(x)$, we can assert these by adding axioms while remaining agnostic about its concrete implementation. Uninterpreted functions (over creating an unconstrained variable for the output) also ensure that if provided with the same arguments, the results are identical. QURE leverages uninterpreted functions to handle third-party calls in user-defined functions for verification as well as to improve extensibility as we discuss subsequently.

3 Overview of QURE

QURE provides a framework for translating UDFs within Spark queries into SQL, using the code translation capabilities of LLMs, and uses formal verification techniques to ensure that LLM-generated SQL is semantically equivalent to the original UDF. Figure 1 depicts the end-to-end workflow of QURE highlighting the key steps.

Figure 2 illustrates the verification workflow for a running example featuring a SQL query (referred to as the *parent query*), labeled Ⓐ, using UDF named 'filter_string', labeled Ⓑ, both of which are input to QURE. QURE first calls a LLM to translate the UDF into SQL, termed *UDF SQL*, (step ①), and then verifies the equivalence between the UDF and SQL (steps ② to ⑤). Upon successful verification, QURE inlines the UDF SQL with the parent query, generating a unified SQL query (referred to as the *final SQL query*) (not shown in the figure). If the translation is unsuccessful,

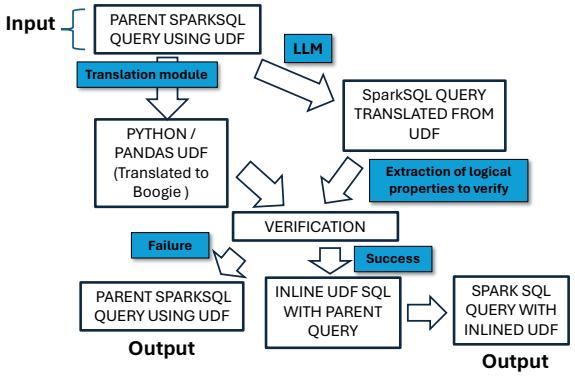

Fig. 1. QURE Architecture

we retain the original parent query, ensuring the correctness of the query's result. We now give an overview of each step in the QURE workflow.

① **Leveraging LLM for UDFs to SQL Translation.** QURE uses LLMs such as *GPT-4o* to translate complex UDFs – including Python Scalar UDFs, Pandas Scalar UDFs, Aggregation UDFs, and Map UDFs – into SparkSQL (see Section 7). For example, the UDF `filter_string` Ⓑ is translated to the LLM(-generated) SQL depicted in Ⓒ. Using LLMs simplifies the translation process and expands the coverage of SQL query synthesis from procedural code, a task that has presented considerable challenges in prior work. The process begins by identifying the UDF type from the query syntax. We have developed few-shot learning prompt templates (capturing instructions and examples of UDF to SQL translations) for each UDF type (described in Section 2.1) which are instantiated with input UDF and relevant metadata from the parent query (e.g., table `lineitem`, '`key`' mapping to the column '`shipmode`') prior to translation. We present the prompt templates for each UDF type in our technical report [9]. Note that while QURE uses an LLM for translation, any other technique can be used to infer SQL without changing the rest of the steps discussed next.

② **Python UDFs to Boogie Translation.** As a necessary step for verifying the equivalence of Python UDFs and the LLM-translated SQL, QURE encodes UDF code into an intermediate verification language, Boogie, through a multi-step translation process. Each step involves custom transformations on the abstract syntax tree (AST) of the Python UDF to translate it into Boogie's syntax while preserving semantics (see Section 5). Boogie can represent complex programming constructs and is suitable for formal verification analysis. Boogie code is subsequently translated by the Boogie back-end into a form suitable for an SMT [10] theorem prover.

$c \in$ constant	:=	*true* | *false* | number_literal | string_literal | list | dictionary
$t \in$ types	:=	*bool* | *int* | *long* | *real* | *str*
$at \in$ arry_types	:=	*list[t]* | *DataFrame* | *Relation* | *Column*
$op \in$ operators	:=	&& | | | | ~ | ! | + | - | * | / | mod | ** | « | » | == | != | < | <= | > | >= | in
$ce \in$ const_expr	:=	c | ce_1 *op* ce_2 | *op* ce
$e \in$ expression	:=	c | var | e_1 *op* e_2 | *op* e | var[ce]
$s \in$ statement	:=	pass | var : t | var = e | if c: s_1 else: s_2 | for e: s | while e: s | return var | function(e_1,…,e_n)

Table 2. An approximate summary of supported Python syntax

To illustrate this process, consider the running example with UDF Ⓑ in Figure 2. The result of the translation to Boogie is shown in Ⓓ, with the results of key transformations highlighted: (a) Boogie is a statically typed language (unlike Python, which is dynamically typed), so types

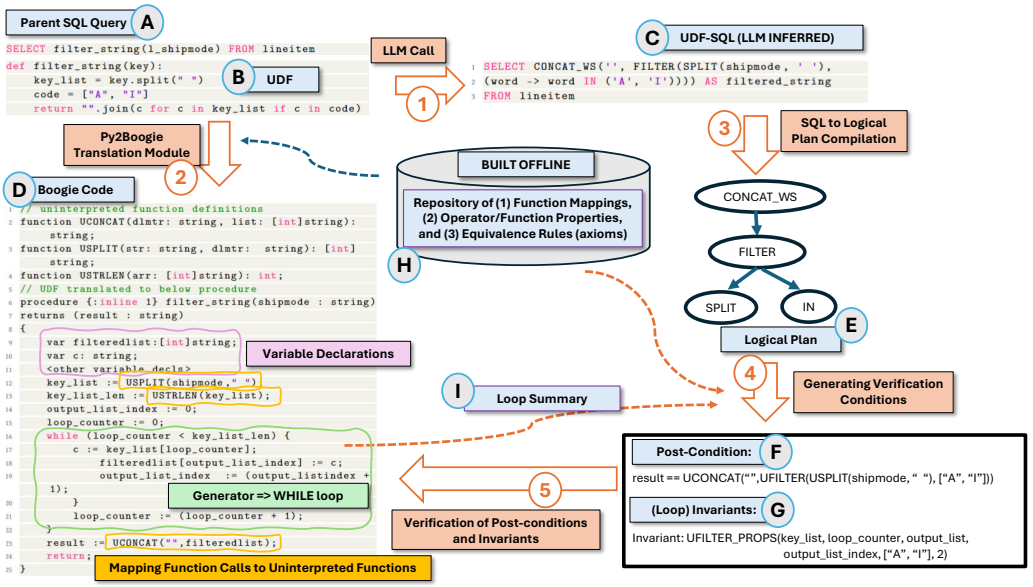

Fig. 2. Illustration of the Verification Workflow: The arrows (numbered from 1 to 5) indicate the key steps in QURE, while the letters (A to I) represent the entities that the steps operate on or produce. The final step that inlines verified UDF SQL with parent SQL is not depicted. Steps 2, 4, and 5 are the focus of this paper.

of all variables need to be declared, (b) the *generator* [2] iterating over members of `key_list` is translated into a `while` loop, including the required loop initialization, counter increment and generator output, and (c) functions in Python like JOIN or SPLIT, that have equivalent functions in SparkSQL, and thus do not require us to explicitly model their semantics, but can be replaced by *uninterpreted functions* such as UCONCAT, USPLIT, that are introduced at the top of the Boogie code. The mapping between Python functions, the equivalen SparkSQL functions and the uninterpreted functions representing them in Boogie is stored in a repository (Ⓗ in Figure 2).

An abbreviated overview of the Python grammar for UDFs supported by QURE is given in Table 2 (note that we do not list functions and constructs introduced via libraries, such as *Pandas* here).

Translation of SQL to Verification Conditions (③ and ④). One of our primary contributions is synthesizing post-conditions and invariants directly from SQL rather than procedural code. Given the inferred SQL, QURE compiles it into a logical query plan [3]. If the SQL is equivalent to the UDF code, the logical plan, represented as an acyclic graph of logical operations, must accurately capture the data processing within the UDF. Each operation corresponds to a segment of the UDF code, and the dependencies between operations must mirror the dependencies between code segments. The mapping between operators and corresponding code segments is automatically explored during verification, except for loops for which we add invariants from operators (discussed in Section 4).

For our running example, Ⓔ depicts a simplified logical plan resulting from the compilation of the query in Ⓒ. The bottom-up flow of operations (built-in Spark functions in this case) – SPLIT, then FILTER (taking as input the output of SPLIT, and a conditional IN expression), and finally CONCAT_WS (taking as input the output of FILTER and a delimiter string) – capture the corresponding expressions in the UDF (Ⓐ), as well as the translated Boogie code (Ⓓ). Note that in this example, FILTER denotes built-in Spark SQL function (see [11]), which is order-preserving.

The logical operations and functions in SQL have well-defined semantics (referred to in QURE as *properties*) and can be precisely modeled offline. These are modeled using uninterpreted functions

(e.g., UCONCAT for CONCAT_WS, USPLIT for SPLIT, UFILTER for FILTER) and stored in a repository (H). QURE traverses the logical plan and generates the post-conditions as a composition (following the same flow defined in the logical plan) of the uninterpreted functions (see (F)). For many functions, such as CONCAT_WS and SPLIT, which are also expressed as functions in the original UDF, detailed modeling of properties is not required. Corresponding uninterpreted functions are generated during the Python-to-Boogie translation and can be matched via name and parameters. However, for each operator with potential procedural implementations inside the UDFs (e.g., while loop capturing the FILTER in (D)), we maintain properties that capture the semantics of the operation (stored in (H)) to be verified. As detailed in Section 4.1, the properties are encapsulated within another uninterpreted function (e.g., UFILTER_PROPS for UFILTER). Through simple syntactic analysis of the Boogie code used in verification (e.g., the WHILE loop in (D)), we can instantiate (e.g., identify parameter values of) the property functions and use them as invariants ((G)) for verification. Note that due to space constraints, Figure 1 does not depict the properties in UFILTER_PROPS, which we list in Section 4.1. Overall, by pre-creating properties for operations and instantiating them via simple syntactic analysis of the procedural code, we reduce the challenge of inferring them directly from procedural code. We discuss in more detail in Section 4.

Verification of Equivalence and Final SQL Generation. QURE builds complete Hoare triples using the post-conditions and invariants sythesized from SQL, and then invokes Boogie's inbuilt verifier to confirm their correctness. Upon successful verification, QURE integrates the LLM SQL with the parent query to generate a single SQL query. This is achieved by adding the UDF SQL as Common Table Expressions (CTEs) using the WITH clause. CTEs function as temporary result sets that are named and can be easily referenced. Note that these CTEs are usually non-recursive and are generally more performant than recursive CTEs.

4 Verification of Equivalence

We consider a user-defined function U, which accepts parameters $P_u = (p_1, p_2, \ldots, p_n)$ and returns a result $R_u = (r_{u_1}, r_{u_2}, \ldots, r_{u_m})$, where each p_i and r_{u_i} are lists of values (scalar values are modeled as singleton lists). Let $p_1, \ldots, p_n$ correspond to columns $c_1, \ldots, c_n$ in an input table T. T itself may result from a subquery over one or more tables, where each c_i may be derived from columns in these tables or correspsonds a constant. In the context of scalar UDFs, T can be viewed as a single-tuple table. Analogous to the notion of tuples in database tables, we use tuples here to denote the list of values across lists in P_u or R_u at the same offset.

A column in a table is modeled as a list, and tables and DataFrames are modeled as lists of lists. The types of parameters passed to UDF can be inferred from the parent query and the database schema. Additionally, in many cases, we can assign an alias type to a variable denoting a table (e.g., type T for Table), and expressions within the UDF that operate on tables/columns can be modeled functionally, with the table name(s) and column name(s) as parameters. This avoids modeling columns of the table, that are not used in any expressions within the UDF. Unless explicitly ordered (via operations/functions within the UDF), we consider a list as unordered (i.e., no defined ordering). Whenever a list is ordered, we maintain additional metadata to denote the ordering. Since lists are represented using arrays in Boogie, we use lists and arrays interchangeably in this paper.

Using the LLM, we translate U to a SQL query S. To aid correct translation, we provide the mappings between the parameter names in U and the column names, as well as the table T (unavailable in the UDF), used to provide the corresponding parameter values, as part of the LLM prompt. Let $R_s = (r_{s_1}, r_{s_2}, \ldots, r_{s_m})$ denote the result of S, where r_{s_i} denotes a column (modeled as a list as mentioned above). We define the semantic equivalence between U and S as follows:

Definition 1 (Semantic Equivalence). *The user-defined function U returning R_u is semantically equivalent to the SQL query S returning R_s if, for any possible table T with columns $(c_1, c_2, c_3, \ldots, c_n)$ passed as $(p_1, p_2, p_3, \ldots, p_n)$ to U, R_u and R_s correspond to an identical bag of tuples. Additionally, R_u and R_s must have the same ordering of tuples if the result ordering is explicitly specified in U.*

Note that when an UDF takes as input or returns ordered tuples (e.g., DataFrames in a Pandas UDF), many database systems (e.g., Spark, Microsoft SQL Server) do not guarantee that the ordering is preserved. In such cases, we consider R_u as unordered. For ease of exposition, we assume in the following that P_u, R_u, and R_s, each consist of a single list, unless otherwise needed.

4.1 Leveraging SQL for Synthesizing Post-conditions and Invariants

We use Hoare-style verification to establish semantic equivalence between U and S, requiring synthesis of pre-conditions, post-conditions and invariants. Here, the pre-conditions specify allowable values for UDF parameters (e.g., multi-tuple parameters cannot be empty) and are easy to infer. Hence, we focus on synthesizing post-conditions and invariants in this section.

Insight. Our observation is that we can compile S into a *logical query plan*, which is an acyclic graph of logical operators and functions.[1] Since the semantics of these operators and functions are well-defined and remain constant regardless of the remainder of the SQL query, we can predefine them offline. We refer to these semantics as *properties*, which are logical assertions that describe how input tuples are transformed into output tuples, as well as the ordering of those output tuples (discussed further in Section 4.3). Consequently, each operation is modeled offline as an uninterpreted function along with its associated properties. The uninterpreted functions for individual operations are then *functionally composed*, mirroring the composition of operators in the logical plan, to derive the post-condition that is used to verify equivalence. When loops are present in the UDF, we use the properties of the corresponding operators (identified by searching through the logical plan) as loop invariants. Overall, this method requires much lighter syntactic analysis of the procedural code in the UDF compared to prior work, as the properties are defined offline and the function compositions are identified through the logical plan.

Example. Our running example in Figure 2 shows the logical query plan/operators (Ⓔ) that the translated query (Ⓒ) is compiled into. The operations CONCAT_WS, SPLIT and FILTER are captured via the uninterpreted functions UCONCAT, USPLIT, and UFILTER defined below (note: in Boogie, `[int]string` denotes a list of strings using `int` values as index).

```
function UCONCAT(delimiter: string, strings: [int]string, len: int): string;
function USPLIT(src: [int]string, delimiter: [int]string, len: int): [int]string;
function UFILTER(srcstr: [int]string,filters: [int]string, filterslen:int):[int]string;
```

The post-conditions are defined by composing these uninterpreted functions, replacing the operators in the logical plan with the corresponding uninterpreted functions by looking them up in the repository (Ⓗ). In our running example (see Ⓕ), this yields:

```
result == UCONCAT("", UFILTER(USPLIT(shipmode, " "), ['A', 'I']));
```

During the translation from Python to Boogie, Python function invocations (like SPLIT and JOIN) within the UDF are also matched with USPLIT and UCONCAT (as discussed in detail in Section 5), ensuring equivalence, provided they share the same function signature. However, for functions like FILTER, which are implemented through lower-level procedural constructs (e.g., conditionals, loops), it is necessary to specify their behaviour for verification, via *properties*. The verifier then confirms the equivalence between the specified properties of the operator and the corresponding implementation within the UDF.

[1]We use "logical operators" and "functions" interchangeably in the context of SQL.

Modeling properties of operations. We model an operator taking input I via the uninterpreted function $F(I)$. To encapsulate the properties $F(I)$ must satisfy, we introduce $F\_PROPS(I, O)$, returning a boolean indicating if the properties $po_1, po_2, \ldots, po_n$ hold for inputs I and outputs O (from some domain) as follows:

$$po_1(I, O) \wedge po_2(I, O) \wedge \ldots \wedge po_n(I, O) \implies F\_PROPS(I, O)$$

If the properties of the function hold, then the result of applying the operator is equal to the function's output

$$F\_PROPS(I, O) \implies O == F(I)$$

These properties functions are also stored in the repository (Ⓗ in Figure 2), and used as loop invariants for operators implemented using loops in the UDFs. During verification, the parameters for the properties function are identified through simple syntactic analysis of the loop (discussed in the next sub-section). This approach simplifies the process of generating verification conditions compared to deriving conditions directly from the code of the loop.

In our running example, the properties of FILTER are captured using an uninterpreted function UFILTER_PROPS:

```
function UFILTER_PROPS(srcstr: [int]string, srclen: int, result: [int]string, resultlen:
    int, filters: [int]string, filterslen: int): bool;
```

with the properties defined (offline) as follows:

- *Property 1 (po_1)*: $0 \leq resultlen \leq srclen$
- *Property 2 (po_2)*: For every character in *srcstr* matching a filter, there exists a corresponding character in the *result*:

$$\forall i \in [0, srclen) : (\exists k \in [0, filterslen) \text{ s.t. } srcstr[i] = filters[k]) \Rightarrow (\exists j \in [0, resultlen) \text{ s.t. } result[j] = srcstr[i])$$

- *Property 3 (po_3)*: Every *result* character comes from the *srcstr*:

$$\forall j \in [0, resultlen) : (\exists i \in [0, srclen) : result[j] = srcstr[i])$$

- *Property 4 (po_4)*: The order of characters matching filters is preserved in the *result*:

$$\forall i, j \in [0, srclen) : i < j \wedge (\exists k, l \in [0, filterslen) : srcstr[i] = filters[k] \wedge srcstr[j] = filters[l]) \Rightarrow (\exists l_1, l_2 \in [0, resultlen) : l_1 < l_2 \wedge result[l_1] = srcstr[i] \wedge result[l_2] = srcstr[j])$$

The parameters for UFILTER_PROPS is identified through syntactic analysis of the while loop (described next) in the UDF (Ⓔ):

```
UFILTER_PROPS(key_list, loop_counter, output_list, output_list_index, ['A', 'I'], 2);
```

4.2 Loop Summaries

QURE performs syntactic analysis to extract *loop summaries* to instantiate parameters of properties function. A loop summary $LS = \{I, O, L, LS_{\text{inner}}\}$ for a loop construct is defined via:

- **Inputs (I):** These are comprised of input arrays $\{I_1, I_2, \ldots, I_k\}$, their lengths $\{n_1, n_2, \ldots, n_k\}$, and their ordering semantics $\{Or_{I_1}, Or_{I_2}, \ldots, Or_{I_k}\}$ (specifying ordered or unordered). An input list is *ordered*, if a function is applied to it that sorts it before the loop, or elements were appended to it from another ordered list By default, an input is unordered.
- **Loop Iteration (L):** These are comprised of an iterator variable i, its initial value i_0, the increment/decrement amount Δi, and the termination condition $T(i, op, B)$ which includes the iterator variable, comparison operator (op), and loop bound (B).
- **Outputs (O):** These are comprised of output arrays $\{O_1, O_2, \ldots, O_l\}$, their lengths $\{m_1, m_2, \ldots, m_l\}$, iterator variables $\{j_1, \ldots, j_l\}$, ordering semantics $\{Or_{O_1}, \ldots, Or_{O_l}\}$, and indicator variables telling whether elements are hashed to an array (for recognizing loops involving dynamic dictionaries/hash tables) $\{h_1, h_2, \ldots, h_l\}$.

- **Inner Loop Summary (LS_{inner}):** Optionally, QURE supports one level of nesting to support operations with nested loops (nested-loop inner-join). A nested loop can be represented similarly to the outer loop using inputs, loop-iteration, and outputs.

Loop summaries are designed to capture the characteristics of common loop constructs that can be translated into SQL. When we encounter a loop that does not fit our model of loop summary (e.g., due to additional or missing variables), we fail the verification and execute the original UDF, ensuring the soundness of our approach.

Ensuring Loop Termination. Loop termination synthesis requires a ranking function (an expression over loop variables) that decreases with each iteration and remains non-negative [41]. For our problem, we handle loops with an iterator variable i, a per-iteration increment or decrement Δi, and a bound B on i. The ranking function R is defined as $R = B - i$ for $\Delta i > 0$, and $R = i - B$ for $\Delta i < 0$. While there can be different ways of using R for verifying loop termination, one option is to add an assertion: $assert(R < R')$ at the end of the loop body (where R' is the value of R at the start of the loop) as well as add an invariant $R \geq 0$.

4.3 Equivalence between Loops and Operators

To establish semantic equivalence between a loop and the corresponding SQL (logical) operator, the properties (i.e., pre-defined logical assertions) of the operators together must completely define the mapping from input lists to output lists in the loop, as well as capture the output lists *ordering* if these are explicitly ordered in the loop.

It is challenging to establish equivalence between arbitrary loop-structures and a SQL operation. However, QURE currently can establish equivalence for loop structures mapping to SQL operations of form: [*OP1*]*OP2* (i.e., sequences that either start with *OP1* followed by *OP2*, or begin directly with *OP2*), with *OP1* and *OP2* being one of the following operations:

OP1 → Filter | Inner-Join
OP2 → Filter | Inner-Join | Row-wise Transformation | Aggregation
Row-wise Transformation → CASE WHEN | Function Calls
Aggregation → Table Aggregation | Group-by Aggregation

In short, *OP1* can be either a Filter or an Inner-Join, while *OP2* may include Filter, Inner-Join, Row-wise Transformations, or Aggregations. The latter encompasses both group-based aggregations as well as table-aggregations (aggregations over the entire table without any grouping of tuples). The sequences of *OP1* followed by *OP2* are modeled as a single (fused) operator in QURE.

QURE supports a fixed set of loop structures, each corresponding to a potential physical operator implementation for a supported logical operation (e.g., hash-aggregation and stream-aggregation for group-by aggregation). These structures may also involve compositions of functions, where each function is registered in the function repository. Using static analysis, QURE ensures that the loop summary for these structures aligns with the expected summaries (one for each possible physical implementation) for $OP1$ or $OP2$. If this alignment is not achieved, verification fails.

We now discuss how we create properties that capture the semantic equivalence between a loop L and (logical) operator OP (of the form [*OP1*] *OP2* as defined above). For ease of exposition, we assume that L processes a single input list I_1 and produces an output list O_1, although our analysis generalizes to multiple lists as inputs or outputs. We define OP as semantically equivalent to L based on the following criteria:

Input to output transformation. The transformation from input I_1 to output O_1 must capture the following:

- *P1: Totality under Constraints:* A tuple r_i in I_1 is either filtered (e.g., using a filtering constraint C captured in OP) or transformed to another tuple s_j in O_1, including the case where multiple

tuples $(r_i, r_k, ..., r_l)$ in I_1 are together transformed to a single tuple in O_1 (e.g., in aggregations). For joins, pairs of tuples from the two tables are considered together.
- *P2: Surjectivity:* No other tuple exists in O_1 that is not the result of a transformation satisfying $P1$.
- *P3: Cardinality:* We verify the handling of duplicate tuples in I_1 as well as the presence or absence of duplicates tuples in O_1, both depending on the operator's semantics. While this property is often implicitly covered under properties P1 and P2, it sometimes requires separate verification.

Ordering consistency. The ordering of tuples in O_1 must be consistent with the ordering of the output from *OP*:
- *P4.1: OP applies explicit ordering on the output, or the output order follows the same order as input:* These are explicitly defined as a property for *OP* and verified over *L*.
- *P4.2: OP results in output that is unordered (i.e., with no defined ordering):* QURE performs static analysis of the loop to ensure there is no explicit ordering within the loop via a function that might reorder tuples, and that the output tuples do not retain the input tuples' order. If either condition is met or the ordering is ambiguous, *L* and *OP* are deemed semantically inequivalent (since *OP* does not guarantee the ordering in the loop).

Note that when properties are used as loop invariants, the input I_1 and output O_1 lists consist solely of tuples processed up to the previous loop iteration, tracked via the input and output array iterators, and there are additional invariants capturing bounds on the size of these variables; we omit them here for clarity.

Examples. To illustrate, we discuss the properties of a few operations below. Note that the examples below only illustrate how to model an operator's behavior (focusing on key aspects) and are not strict logical representations. In practice, the number and exact formulation of properties may vary based on strictness requirements, including factors such as null checks, floating-point precision, ordering. These additional checks can be incorporated (or removed) into an operator's modeling without altering the rest of the QURE framework.

Row-wise Transformations. Each input yields a unique output by applying a transformation function *F* (modeled as uninterpreted function) or using IF-ELSE statements, (r_i denotes tuple at index i in I_1, $\exists! s_j$ denotes there exists a single output tuple s_j)

Totality and Surjectivity: $\forall r_i \in I_1, \exists! s_j \in O_1$ s.t. $s_j = F(r_i)$

Filter. Outputs are determined based on a specified condition *C*. Filter conditions are expressed as functions for strict matching, thereby avoiding weaker invariants. Certain variants of Filter (e.g., the example in Figure 2) output tuples in the same order as input.

Totality (w/ constraints) and Surjectivity: $\forall r_i \in I_1, \exists! s_j \in O_1$ s.t. $(C(r_i) \Leftrightarrow s_j = r_i)$
Cardinality: $|\{s_j \in O_1 \mid s_j = r_i\}| = |\{r_i \in I_1 \mid C(r_i) \wedge s_j = r_i\}|$
Ordering: If $r_i, r_j \in I_1 \wedge i < j \wedge C(r_i) \wedge C(r_j)$, then $\exists s_k, s_l \in O_1$ s.t. $k < l \wedge s_k = r_i \wedge s_l = r_j$

Group-by Aggregates. Group-by aggregates come in two variants: hashing-based and streaming-based, which can be distinguished by examining their loop structure.

(A) Hash-Aggregate. Hash-aggregate involves hashing and look-up operations. Here, we assume the input values are grouped on a *key* column, and values of *val* column for each group is aggregated using a function *A*. $Keys(I_1)$ is shorthand for all the keys in I_1:

Totality: $\forall k \in Keys(I_1), \exists! s_j \in O_1$ s.t. $s_j.key = k$
Surjectivity: $\forall s_j \in O_1, \exists k \in Keys(I_1)$ s.t. $s_j.key = k \wedge s_j.val = A(\{r_i.val \mid r_i \in I_1 \text{ and } r_i.key = k\})$

(B) Stream-Aggregate: In this variant, the input is already sorted by key column. In addition to the properties listed above under hash-aggregates, keys in the output are also sorted:

Ordering: $\forall r_i, r_j \in I_1$ s.t. $r_i.\text{key} \neq r_j.\text{key} \wedge i < j, \exists! s_k, s_l \in O_1,$
s.t. $s_k.\text{key} = r_i.\text{key} \wedge s_l.\text{key} = r_j.\text{key} \wedge k < l$

Table Aggregates. A single output tuple is generated by aggregating over a set of input values using an aggregation function A. Below, instead of two input lists, we assume each tuple in I_1 has two values key and val for simplicity.

Totality and Surjectivity: $\forall s_j \in O_1, s_j.\text{val} = A(\{r_i.\text{val} \mid r_i \in I_1\})$
Cardinality: $|O_1| = 1$

Inner-Join. Similar to aggregates, the types of inner joins (nested-loop, merge join, and hash join) can also be inferred from the loop structure. We discuss the properties of nested-loop in detail and outline how similar properties can be defined for other types of joins.

(A) Nested-Loop. In a nested-loop join, each tuple r_i from one input I_1 is processed in the outer loop, while the inner loop iterates over every tuple r_j from the other input I_2, evaluating the join condition $C(r_i, r_j)$. If the join condition is met, the tuples are combined into a new tuple s_a using a function $J(r_i, r_j)$ and added to the output O_1.

Outer Loop:

Totality and Surjectivity: $\forall (r_i, r_j) \in I_1 \times I_2, s_a \in O_1 : (s_a = J(r_i, r_j)) \Leftrightarrow C(r_i, r_j)$;
Cardinality: $|\{s_a \in O_1 \mid s_a = J(r_i, r_j)\}| = |\{(r_k, r_l) \in I_1 \times I_2 \mid r_k = r_i \wedge r_l = r_j\}|$
Ordering: $\forall (r_i, r_j), (r_k, r_l) \in I_1 \times I_2, (s_a, s_b) \in O_1, C(r_i, r_k), C(r_j, r_l), s_a = J(r_i, r_k), s_b = J(r_j, r_l) :$
$(((i < k) \vee (i = k \wedge j < l)) \Rightarrow (a < b)) \wedge (((i > k) \vee (i = k \wedge j > l)) \Rightarrow (b > a))$

Inner Loop: Let r_i be the tuple from the outer loop I_1.

Totality and Surjectivity: $\forall r_j \in I_2, s_a \in O_1 : (s_a = J(r_i, r_j)) \Leftrightarrow C(r_i, r_j)$;
Ordering is similar to outer loop order with r_i fixed.

(B) Merge Join. In merge join, both input lists are pre-sorted and tuples from the lists with matching conditions are merged by moving through each list in parallel. Thus, the loop here consists of two iterators one on each list.

Totality and Surjectivity: $\forall (r_i, r_j) \in I_1 \times I_2, (s_a) \in O_1 \times O2) : (s_a = J(r_i, r_j)) \Leftrightarrow C(r_i, r_j)$;
Ordering is similar to outer loop ordering in nested loops.

(C) Hash Join. A hash join first constructs a hash table from the one input, then probes it to find matches. The first step is similar to group-by aggregates, with the aggregation collecting a set of values based on keys. The probe can be modeled as a nested-loop where the outer loop matches the key in the hash table, and the inner loop joins all values with matching keys.

Filtered Group-by Aggregates. Here, we apply a filter condition before aggregating the input values.

$\forall r_i \in I_1$ s.t. $C(r_i), \exists! s_j \in O_1$ s.t. $s_j.\text{key} = r_i.\text{key}$
$\forall s_j \in O_1, s_j.\text{val} = A(\{r_i.\text{val} \mid r_i \in I_1 \wedge r_i.\text{key} = s_j.\text{key} \wedge C(r_i)\})$

The filtered versions of other operations (e.g., filtered joins) can also be similarly written. Likewise, inner-join followed by an aggregates is akin to filtered aggregates but include the inner-join condition $C(r_i, r_j)$ applied to both tables.

Examples of Boogie code modeling lower-level constructs, such as hash-tables, look-up operations on hash-tables, as well as loop-invariants can be found in Appendix.

4.4 Limitations in Loop Handling

QURE restricts loop coverage to operators of the form $[OP1]OP2$ due to the inherent complexity of verifying semantic equivalence between loops and their corresponding SQL operators. For these operators, properties can be reliably checked to prevent incorrectly declaring equivalence between loops and operators when they are not equivalent. Such errors may arise when the verified properties only partially capture the loop's behavior. Below we discuss some of the additional limitations related to loop structures.

- *Pre-Loop and Post-Loop Code Segments:* QURE assumes that the logic for input to output mapping (beyond initialization of iterator variables and inputs/outputs) as well as ordering is solely confined

within the loop body, and thus, can be checked via loop invariants. This assumption implies that operators with any semantics not solely encapsulated within the loop's body are not covered.

- *Multiple Loops Mapping to a Single Operator:* Operations expressed via multiple loops are not captured, except for the scenarios discussed earlier (e.g., variations of inner joins). The challenge lies in identifying precise invariants for each loop that covers only part of the operator's semantics.
- *Multiple Operators for a Single Loop:* QURE currently only handles fused versions of basic operations such as filters, joins, and aggregations, but not other uncharacterized operator combinations.

4.5 Discussion

Handling changes in dataflows. In some cases, the query optimizer may generate a logical plan with an operator ordering that differs from the execution flow in the corresponding UDF code segments. For instance, the logical plan might specify a flow $A \rightarrow B \rightarrow C$, while the UDF code segments implement the flow as $A \rightarrow C \rightarrow B$, such as in cases involving commutative operators. To address such scenarios, we introduce equivalence rules that help explore post-conditions and invariants across different implementations flows within the plan. For example, global equivalence rules, such as the one below, can assist the verifier in proving equivalence across various combinations of operator (here, the prefix `U` denotes uninterpreted function modeling the logical operation):

$$\text{UStreamAgg}(\text{USort}(R, G1), G1, \text{UAggFunc}(A)) \equiv \text{USort}(\text{UHashAgg}(R, G1, \text{UAggFunc}(A)), G1)$$

By leveraging axioms that define these equivalence rules, the verifier can reason about the correctness of post-conditions and invariants, even when the translated post-conditions and invariants do not perfectly align with the UDF code segments. While QURE currently supports a limited set of such equivalence rules, these rules have been extensively studied in the context of query optimization in relational databases, allowing us to reuse them for well-known SQL operators.

Why not use the physical plan from the query optimizer? Database query optimizers apply equivalence rules to transform logical plans to generate physical plans that might provide the best performance. However, physical operators providing the best performance might not be the ones used in the UDF. In many cases, the candidate physical implementation can be identified via lightweight analysis of the loop as discussed earlier. However, when the implementation is unclear, QURE explores each physical implementation for the logical operator in the plan for verification.

5 Modeling Python in Boogie

As outlined in Section 3, verifying equivalence between UDFs and translated SQLs requires expressing the semantics of UDF in Boogie, an intermediate verification language. The translation is challenging for a number of reasons: (1) Python offers a significantly richer variety of syntactic constructs (e.g., dictionaries, generators and additional loop types) than Boogie. (2) Python UDFs commonly invoke functions from libraries such as *NumPy* [4] and *Pandas* [6] for which the source-code is not available, or sufficiently complex to make the verification itself challenging. Thus, most existing approaches for UDF translation do not handle library functions. However, as the rich set of libraries available is one of the main reasons for Python use in UDFs, translations framework should be able to support library functions, and do so without requiring significant re-writes. (3) A key abstraction in Pandas UDFs are *DataFrames* [13]. Basic operations over DataFrames have different semantics than the same operations for other variable types (e.g., *addition* of two DataFrames effectively introduces a loop that iterates over all rows in the referenced columns), and can introduce relational abstractions (e.g., through the `.groupby()` function [7]). We illustrate how we address these challenges in the following.

5.1 The AST Transformation Framework

QURE has a custom Python-to-Boogie translation component (see step ② in Figure 2) which applies a series of *transformations* on the Python AST [59] of the UDF to translate it to semantically equivalent Boogie. Each transformation is defined as a rewrite rule operating on the Python AST that is triggered when one or more *pre-conditions* are satisfied. Note that these pre-conditions can involve the entirety of the AST (e.g., a pre-condition can require that a variable be assigned to only once). These transformation include: (1) converting lists and dictionaries into Boogie-compatible arrays, (2) converting different loop types into Boogie-compatible `while` loops with explicit conditions and iterators, (3) rewriting generator expressions and list comprehension as loops with equivalent iterators, (4) translating function and method calls, including library functions, into Boogie uninterpreted function calls with relevant axioms, (5) translating DataFrame operations either into equivalent Python code (and, subsequently, through other transformations, to Boogie code), or modeling relational operators through equivalent uninterpreted functions over special *Relation* and *Column* types, and (6) mapping functions on data types not native to Boogie (e.g., *DateTime*) using uninterpreted functions.

To illustrate the transformation framework, consider a simple UDF, which executes a *dictionary* lookup for values in the `index`, and returns `-1` otherwise:

```
def udf_lookup(key):
    index = { "AIR": 0, "MAIL": 1}
    if key in index:
        return index[key]
    else:
        return -1
```

The semantics of the predicate (`key in index`) and `index[key]` lookup is modeled via a sequence of 3 transformations. The first transformation is invoked only if a *pre-condition* holds that requires that `index` acts as a constant, i.e., it is assigned to once, we never invoke a function that modifies it, and no reference to `index` is passed. If the pre-condition holds, the transformation replaces all references to `index` with the value assigned, resulting in (note that, while all transformations modify the AST, we show, for clarity, the corresponding Python code):

```
def udf_lookup(key):
    if key in {'AIR': 0, 'MAIL': 1}:
        return {'AIR': 0, 'MAIL': 1}[key]
    else:
        return -1
```

The second transformation replaces expressions checking whether a key is `IN` a constant dictionary by a disjunct of equality checks:

```
def udf_lookup(key):
    if (key == 'AIR' or key == 'MAIL'):
        return {'AIR': 0, 'MAIL': 1}[key]
    else:
        return -1
```

The third transformation replaces key-lookups against a constant dictionary with a function performing the same logic, again using equality checks. The code below shows the *Boogie code* resulting from this transformation, type inference and the final translation:

```
Procedure dict0(key: string) returns (value: int) {
     if ( key == "AIR")  { value := 0; return; }
     if ( key == "MAIL")  { value := 1; return; }
}
Procedure udf_lookup(key:string) returns (result:int) {
   var boogie_value_dict0: int;
   if (key == "AIR" || key == "MAIL")  {
      call boogie_value_dict0 := dict0(key);
      result := boogie_value_dict0;
      return;
   }
   else {  result := -1;  return; }
```

```
}
```

Modeling Dynamic Dictionaries. Dictionaries with dynamic inserts and updates are modeled using two arrays: one for storing keys and the other for values. The key idea involves modeling the lookup function using axioms that specify the behavior when a key is present or absent without providing an explicit implementation. If the key is found, the function returns its index; if not, it returns the length of the array, indicating that the key is not present.

```
type KeyValue;
function getKey(kv: KeyValue): int;
function getValue(kv: KeyValue): int;
function lookup(ad: [int]int, k: int, len: int) returns (int);
axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
        (0 <= lookup(ad, k, len) && lookup(ad, k, len) < len) ==> (getKey(data[lookup(ad,
      k, len)]) == k));

axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
        (lookup(ad, k, len) == len) <==> (forall i: int :: (0 <= i && i < len) ==> (
     getKey(data[i]) != k)));

axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
         len == 0 ==> (lookup(ad, k, len) == -1));

axiom (forall ad: [int]int, k1: int, k2: int, len: int ::
        (0 <= lookup(ad, k1, len) && 0 <= lookup(ad, k2, len) && k1 != k2) ==> (lookup(ad
     , k1, len) != lookup(ad, k2, len)));

axiom (forall ad: [int]int, k1: int, k2: int, len: int, len2: int ::
        (0 <= lookup(ad, k1, len) && 0 <= lookup(ad, k2, len2) && len <= len2) ==> (
     lookup(ad, k1, len) != lookup(ad, k2, len2)));

axiom (forall ad: [int]int, k: int, len: int, len2: int ::
        (0 <= lookup(ad, k, len) && 0 <= lookup(ad, k, len2) && len <= len2) ==> (lookup(
     ad, k, len) == lookup(ad, k, len2)));

procedure udf(data: [int]KeyValue, data_len: int) returns (keys_array: [int]int,
     values_array: [int]int, ad_len: int)
requires data_len > 0;
{
    var i, j, k, v: int;
    ad_len := 0;
    i := 0;
while (i < data_len)
    {
        k := getKey(data[i]);
        v := getValue(data[i]);
        j := lookup(keys_array, k, ad_len);

        if (j == ad_len){
          keys_array[ad_len] := k;
          values_array[ad_len] := v;
          ad_len := ad_len + 1;
        }
        else{
         values_array[j] := values_array[j] + v;
        }
        i := i + 1;
    }
}
```

Type Inference. Since Boogie is *statically* typed, unlike Python, the translator needs to infer types during the translation. Here, the types of input parameters to the UDF are inferred by looking up the types of corresponding columns from the database schema. The types for variables instantiated within the UDF are inferred using rules based on the operations and the types of columns on the right side of the assignment. The return types of functions can be inferred from the function repository. If the type of any variable is ambiguous, the verification is considered unsuccessful.

5.2 Handling Function Calls

For many commonly used library functions in Python UDFs, there exist equivalent functions in SQL, which are (typically) correctly identified by LLMs in their UDF translation. To extend the verification framework to such functions for which no source code is available, QURE uses a *function repository* to track the required equivalencies between Python functions and SQL functions, as well as to capture any additional semantics required for the verification. QURE currently supports more than 130 frequently used Python and DataFrame functions (see Table 3 for a detailed breakdown).

Table 3. Functions in the QURE repository

Type of function	Number
Mathematical expressions	12
String manipulation and counting	17
Type manipulation and special types (e.g., *DateTime*)	11
Pandas DataFrame specific functions	46
Relational operators	44
Other	2

Concretely, this repository captures the following information for each pair of equivalent Python and SQL function:

Py_func : Python function, parameters and their types, return type of the function
SQL_func : Equivalent SQL function, its parameters and their types
b_func : Boogie *uninterpreted function* representing *Py_func*
b_code : Additional code required to encode *b_func* in Boogie
axioms : If needed, any additional *axioms* required for verification, e.g., encoding *commutativity*

Because Python is dynamically typed, and Boogie requires static type declarations associated with each function, the repository may contain multiple entries of the Python function, each of which is associated with different parameter types.

Using the repository, QURE (a) can map pairs of equivalent functions in UDF/SQL to a canonical uninterpreted function in the Boogie code, (b) use axioms to capture necessary or additional semantics for verification, (c) support composability of functions with other expressions in UDF via generated Boogie code (which can be multiple lines in order to capture input and output variables). Since the specification of functions itself is concise, additional library functions with equivalent SQL can be registered easily and allow for extensibility of QURE without requiring code changes.

5.3 Modeling DataFrames

Since DataFrames are not native to Boogie, verifying UDFs with DataFrames requires modeling their semantics. A DataFrame passed as input to a UDF corresponds to a table wherein column labels match the table's column names. Each column in a DataFrame is modeled as a Boogie array, e.g., a reference `df["lquantity"]` to a column *lquantity* in a Dataframe *df* is translated to a Boogie array *lquantity*. When the UDF does not specify the columns in a DataFrame columns explicitly, QURE uses the parent SQL query and SQL metadata to infer them. For each column, we also add a variable representing the number of rows, allowing us to translate row iterators into offset-based iterators (see example below). Column types are inferred using the process described earlier.

Operations within DataFrames — such as column manipulations, aggregations, filtering, and transformations — are translated into a equivalent set of Python instructions, which can then be further translated into Boogie or represented by uninterpreted functions (e.g., `df.filter(...)` and `df.groupby(...)` for relational operations, as well as other functions when full implementation is not required). Additionally, we can often assign an alias type to a table or DataFrame (e.g., type `T` for a table). Expressions within UDFs that operate on these tables are modeled functionally, using the table name and column names or expressions as parameters. For instance, `sort(R1, "A",`

"ASC") represents a sorting operation on Table R1 by column A in ascending order, where R1 can be defined as type T, without needing to model other columns in $R1$. Similarly, since input columns and their types are inferred from the parent query and database schema, DataFrame functions operating on these columns within a UDF can use a generic type alias, TCol, for these columns, instead of defining for each possible data type (e.g., int, real, string).

Example:

```
def udf_0(df):
    taxmax = -1
    for _, row in df.iterrows():
        taxmax = max(taxmax, row["l_quantity"] * row["l_discount"] * row["l_tax"])
    return pd.DataFrame({"l_taxmax": [taxmax]})
```

Here, a first transformation explicitly maps all referenced columns (e.g., df['l_tax']) to individual variables (e.g., df_l_tax) representing the column; for the purpose of subsequent Boogie translation, these are also added as parameters to the Boogie representation of the UDF (since df is an input parameter). A second transformation changes the DataFrame iterator row into an offset i_df0 used to track row offsets. Finally, a third transformation changes the FOR loop (not supported in Boogie) to WHILE, including initialization and increment of i_df0. The resulting Python code (before subsequent translation to Boogie) is shown below:

```
def udf_0(df_l_discount, df_l_quantity, df_l_tax, df_len):
    taxmax = -1
    i_df0 = 0
    while i_df0 < df_len:
        taxmax = max(taxmax, df_l_quantity[i_df0]  * df_l_discount[i_df0]
                     * df_l_tax[i_df0])
        i_df0 = i_df0 + 1
    return taxmax
```

6 Inlining UDF SQL Using CTEs

For inlining UDF SQL with the rest of the SQL, we place the UDF SQL as a common table expression (CTE) created using the WITH clause. CTEs serve as temporary named result sets that can be referenced within the SQL query. Consider a simple UDF designed to filter numerical values:

```
def filter(num):
    return num < 7
SELECT filter(num) FROM tablex
```

The UDF SQL can be inlined with a CTE, as shown below:

```
WITH tempTable AS (
  SELECT (l_quantity < 7) AS l_cond FROM tablex )
SELECT l_cond FROM tempTable
```

When UDFs involve more sophisticated logic, such as aggregations or window functions, we need to accurately reflect these in the CTE. However, the UDF itself might not specify grouping or ordering directives—these are often declared in the parent query. To ensure the correctness of the final query, these attributes must be integrated into the CTE.

There are two approaches to this integration:

- Translate the UDF to SQL without specifying grouping or ordering attributes: Initially, the UDF is translated assuming aggregation over the entire input table. This SQL representation is then verified. Post-verification, the necessary grouping or ordering attributes from the outer query are incorporated into the UDF's SQL representation within the CTE.
- Incorporate grouping or ordering attributes directly into the UDF translation: By providing these attributes as additional context, the UDF's SQL representation can be crafted with the correct grouping or ordering from the outset.

As an example, consider an UDF that standardizes values by subtracting the mean and dividing by the standard deviation:

```
def udf(df):
    return (df - df.mean()) / df.std()
```

which is used in a query grouped by date attributes:

```
SELECT explode(collect_list(udf(l_quantity)))
FROM lineitem
GROUP BY year(l_shipdate), month(l_shipdate)
```

The inlined version incorporating this UDF through a CTE is as follows:

```
WITH t AS (
  SELECT *,
    (l_quantity - mean(l_quantity) OVER
    (PARTITION BY year(l_shipdate), month(l_shipdate))) /
    std(l_quantity) OVER (PARTITION BY year(l_shipdate), month(l_shipdate))
    AS udf_0_col
  FROM lineitem
)
SELECT explode(collect_list(udf_0_col)) FROM t
GROUP BY year(l_shipdate), month(l_shipdate)
```

In this example, the CTE `t` incorporates the window function logic, applying the UDF within the context of each partition defined by `year(lshipdate)` and `month(lshipdate)`. The outer query then uses the results of this computation.

By representing UDFs as CTEs, SQL queries can maintain simplicity and readability without sacrificing performance. This approach allows for the clear expression of complex logic and facilitates the integration of advanced data transformations into analytical queries, providing a scalable and maintainable solution for working with UDFs in SQL environments.

7 Empirical Evaluation

UDF Benchmark. We evaluated QURE over a diverse set of UDFs (summarized in Table 4), targeting both generic Python as well as different variations of Pandas UDFs (see the UDF categories in Section 2.1). While UDFs for the categories *Python* and *Pandas-Scalar*, *Pandas-Agg*, and *Pandas-Map* were are collected by us (from sources such as *GitHub* and *Stack Overflow* to ensure their practical relevance), the benchmark workloads also include 22 *TPC-H queries* written using *Pandas*, similar to the ones used in PyTond [57]. We also have 9 *Python-Scalar* UDFs based on UDFs shared in Froid [51], as well as 10 UDFs based on notebooks shared by *PyFroid* [21]. Because the data for the original UDFs are largely not available, all UDFs are modeled to use the TPC-H schema, allowing us to execute them and measure performance improvements. For the UDFs we collected, our goal was to evaluate QURE's capability to translate and verify UDFs. Therefore, we prioritized selecting UDFs where a complete SQL-equivalent translation might exist (although equivalent SQL translations do not exist for some of them), even if the LLM may not always correctly generate it. Concretely, this led us to filter out a large number of UDFs containing ML invocations or graphics/plotting functions, which clearly lack SparkSQL equivalents. The benchmark can be found here [9].

Ground-Truth. To validate the semantic equivalence of a UDF and the SQL inferred by the LLM, we used both manual inspection of the code as well as execution of the queries containing the UDFs alongside the UDF-inlined queries to confirm identical results on a TPC-H dataset. Manually verified queries served as the ground-truth for evaluating the automated verification used in QURE.

Table 4. Workload characteristics and their impact on translation/verification

Category	# queries	# lines (min, avg, median, max)	# operations (min, avg, median, max)	# UDFs with loops	#queries translated correctly (*GPT-4o*)	#queries w/ correct LLM-translations verified	#queries verified when given correct translations
Python	27	(4, 19, 18, 42)	(3,23,14,129)	6	24	24	24
Pandas-Scalar	19	(3,12,10,38)	(2,11,8,32)	7	15	15	15
Pandas-Agg	23	(3,12,13,27)	(1,7,6,22)	0	18	18	20
Pandas-Map	15	(3,10,9,26)	(2,9,5,35)	8	11	10	12
TPC-H (Pandas)	22	(4,11,10,25)	(5,10,10,25)	0	21	21	22
Froid-based	9	(1,7,8,15)	(1,7,8,12)	0	9	9	9
PyFroid-based	10	(3,12,11,38)	(1,10,7,35)	0	8	8	8
Total	125	(1,16,15,42)	(1,10,10,129)	21	106	105	110

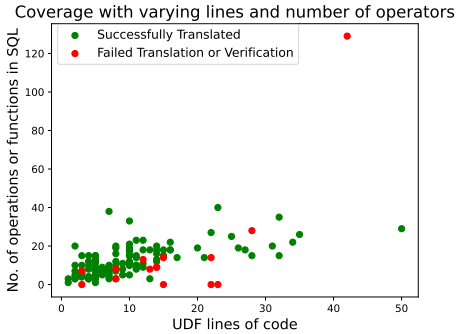

Fig. 3. Coverage of UDFs with varying lines of code and operations in the translated SQL. The value 0 on the Y axis denotes that no translation returned by the LLM.

Performance Evaluation. For performance evaluation, we use HDInsight (HDI) clusters (version 5.1, US East 2 region), using Apache Spark 3.3 on E8 V3 nodes, featuring 8 cores and 64 GB RAM. We vary the number of nodes between 1 and 12, and the TPC-H scale factor between 10 and 100.

7.1 Coverage of UDFs

Table 4 depicts the accuracy of translations performed by LLM (GPT-4o) that we audited manually and by execution over the TPC-H dataset of scale factor 10. The table also shows which fraction of queries in each category were successfully verified by QURE; here, we differentiate between the case (a) where LLM translation was correct, as well as (b) where we added correct SQL translations to replace LLM mistranslation (where needed).

QURE's coverage. QURE can verify equivalence between UDFs and SQL for 110 (88%) queries when given correct translations and 105 queries (84.8%) when not correcting LLM mistakes. As depicted in Figure 3, QURE achieves a high verification accuracy across UDFs with varying lengths (between 1 to 40 lines of code) and number of SQL operations[2].

One reason for the high coverage of QURE is the large number of procedural constructs modeled in QURE. QURE captures conditional logic (e.g., IF-ELSE statements) and list-based operations within UDFs (similar to prior work) while also covering UDFs involving dictionaries and look-ups (lower support in prior work). QURE covers a substantial fraction of UDFs with string manipulations, function calls along with their composition with other operations, and DataFrame operations, all common in data transformation and preparation tasks but with limited support in prior work. QURE also provides coverage of generator expressions [2], supporting the use of anonymous functions within UDFs. Many loops in Spark UDFs map to row-wise transformations, aggregations, and filter operations in Spark, and QURE successfully verifies 18 of the 21 UDFs with such loops. QURE can establish semantic equivalence between loops and operations such as filters, table aggregations,

[2]Here, we count each operator, a function call or comparison as 1 operation.

group-by aggregations, inner joins (nested loop, hash join, and merge join), and also extends to their combinations. Table 5 lists the types of operations for which QURE can establish equivalence.

Besides the volume of procedural constructs and functions, high coverage is facilitated by *the core idea of QURE, namely, to use the logical plan (or query tree) of the SQL translation to functionally compose the verification conditions* (see Section 4 for details). Note that verification conditions for an operation or function remain the same regardless of the length or complexity of the UDF. The structure (data flow in the logical plan) helps QURE compose the verification conditions across all operations in the UDF. As a result, the verification coverage of QURE is not very (anti-)correlated with UDF length or number of operations (at least for the scale and complexity seen in Apache Spark UDFs). Instead, QURE's coverage is more dependent on existence of specific constructs that can or cannot (e.g., yield operations, no explicit ordering specified in UDF while the SQL operator assumes one, or missing else parts in conditional statements) be logically modeled, or loops that do not belong to the loop patterns we cover (see Section 4.2).

Translation Accuracy of LLMs. We note that *GPT-4o* outperforms GPT-4 and GPT 3.5 models for our translation task. As a key feature, QURE can leverage improvements in translation accuracy due to newer LLMs without changes to QURE itself. We observe that LLMs excel in capturing loops and conditional logic and can map functions, such as string manipulation, to equivalent SQL, particularly for operations well-defined within both Python and SQL environments. LLMs are also effective at translating column-based operations and aggregations, which align closely with operations inherent to SQL. Among the different categories of UDFs, *Pandas-Agg* UDFs generally use simpler, native Pandas API functions, including basic aggregation functions like `sum()`, `mean()`, or `max()`, which have direct SQL counterparts. Therefore, LLMs demonstrate the highest accuracy for such UDFs in this category. In contrast, *Python*, *Pandas-Scalar*, and *Pandas-Map* UDFs often incorporate more complex logic, leading to lower LLM accuracy. A significant portion of errors arises from incorrect function-to-SQL mappings, especially when SQL lacks direct function equivalents; here, GPT-4o performs signifcantly better than prior LLMS. Other errors include incorrect ordering of operations, overly complicated UDF logic, and omission of functions during translation.

Table 5. Coverage of loops corresponding to SQL (logical) operations. QURE supports a fixed set of loop structures, each corresponding to a potential physical operator implementation for a supported logical operation (e.g., hash-aggregation and stream-aggregation for group-by aggregation).

Operations	Popularity in Spark UDFs	Prior Work([17, 51])
Compositions of Functions and SQL Operations	High	No
Projection / Row-wise Column Transformation	High	Yes
Filtering	High	Yes
Table Aggregation	High	Partial
Group-by Aggregation	Medium	No
Filtered Aggregations	Medium	Partial
Joins (Nested Loop, Hash, Merge)	Unseen	Nested Loops Only
Filtered Joins	Unseen	Nested Loops with Filters
Join with Aggregation	Unseen	Min/Max with Nested Loop

7.2 Performance Improvement

Table 6 depicts the performance improvement after UDF inlining for each category of the UDFs. Among the UDFs that QURE can verify, UDF inlining results in median performance gains of 23.7× in single-node configurations and 12.5× in configurations utilizing 12-cluster nodes. We analyze the performance based on various factors.

Impact on different types of UDFs. Our analysis of a sample of Python UDFs shows that about 20-30% of the time is spent on data movement and rest on computation. After inlining the UDF, the data movement overhead is completely eliminated and the computational overhead is reduced by 90%

Table 6. Speed-up of UDF-inlined queries w.r.t original queries. At a scale factor of 100, the original queries for Pandas Map run out of memory (OOM), so no speed-up is reported.

	Speed-up (minimum, average, median, maximum)			
	1 node		12 nodes	
	SF = 10	SF = 100	SF = 10	SF = 100
Python	1.1, 20.6, 20.02, 52	0.76, 35.73, 39.49, 102	0.08, 11.2, 10.8, 24.1	0.05, 14.3, 16.64, 43.72
Pandas-Scalar	0.10, 6.4, 5.5, 16.4	0.02, 5.10, 6.53, 9.7	0.23, 7.98, 8.24, 12.70	0.11, 4.29, 5.94, 8.78
Pandas-Agg	0.96, 10.28, 11.31, 31.51	0.65, 11.3, 9.75, 37.73	0.70, 12.94, 9.89, 21.55	.54, 6.84, 7.6, 22.4
Pandas-Map	21.29, 34.14, 25.22, 157.22	OOM for non-inlined queries	12.0, 41.54, 12.26, 212.12	OOM for non-inlined queries
Froid-based	1.06, 79.23, 83.07, 124.33	1.2, 272.31, 162.97, 327.96	0.85, 37.52, 42.64, 70.89	1.2, 116.86, 93.34, 319.98
PyFroid-based	8.17, 10.21, 24.78, 75.94	0.95, 13.92, 15.91, 23.79	1.51, 22.29, 33.91, 49.61	2.42, 18.40, 15.56, 30.72

for Python UDFs. This results in nearly an order of magnitude improvement in performance for Python UDFs on a single node and small datasets (10 GB). However, Pandas UDFs benefit from vectorization (generally one third faster than Python UDFs for small datasets). Inlining reduces data movement overhead completely and computational overhead by 2× to 5× for most queries. An exception is seen in Pandas Map UDFs, where data movement overhead is more pronounced as the JVM communicates with the Python worker through tables, showing better speedup due to reduction in data movement.

Impact of large datasets and reduction in Out of Memory (OOM) errors. As the dataset size increases, both data movement overhead and computation time increase, widening the performance gap between inlined and non-inlined queries. We observe speed-up of 5× to 10× for SF = 100 compared to SF = 10. The results are more pronounced for queries with Pandas Map UDFs which run OOM for SF = 100. With inlining, we see a reduction in such OOM errors.

Impact of parallelism. As the number of nodes is increased from 1 to 12, the performance of queries with UDF improve by 2× to 3× more compared to the corresponding inlined versions due to the proportional reduction in data and computation overheads. However, parallelization does not always improve the performance of UDFs, as data transfer overheads start to dominate for some queries with increasing nodes.

Performance regressions. A few UDFs exhibit performance regressions after inlining. Most of such queries includes Pandas operations, which benefit from vectorization, but their equivalent window-based SQL queries run more slowly. Additionally, there are two non-Pandas queries where string-based data manipulation functions are much faster than their equivalent SQL functions. Moreover, some queries only exhibit regressions for specific numbers of nodes in the cluster, indicating a sweet spot where data transfer and computation overheads are minimized for the UDFs.

7.3 Comparison with *ByePy*

Since, to the best of our knowledge, among the competing techniques, only ByePy is openly available for testing (via the online interface at `https://apfel-db.cs.uni-tuebingen.de/`), we compare the performance of UDFs translated by ByePy with those translated by QURE on PostgreSQL, using UDFs that do not contain embedded SQL (thereby ruling out the queries in the ByePy benchmark), as QURE can not cover these.

Setup: We selected 21 UDFs from our benchmark that (a) ByPye is able to translate and (b) QURE is able to translate and verify. To ensure a representative sample, this workload contains 3+ queries from each of the UDF categories defined in Section 2.1. Since ByPye input and output are in PostgreSQL format, we translated the original UDFs by (1) adapting them to use PostgreSQL syntax and functions and (2), for UDFs that use DataFrames (which PostgreSQL does not support), translating the corresponding DataFrame columns to arrays and functions to loops over the DataFrame(s) performing the same computation; for the QURE translations, we had the LLM produce SQL targeted at PostgreSQL as well. All experiments were then conducted on PostgreSQL

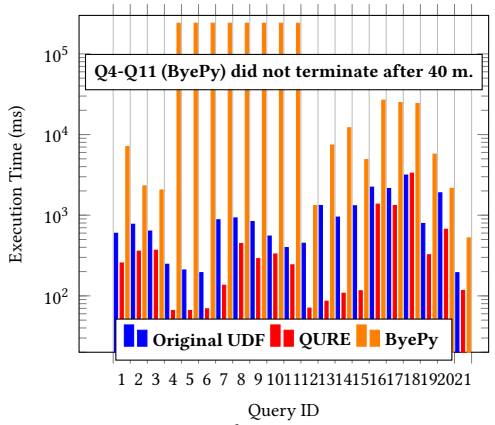

Fig. 4. ByePy performance comparison

16.3, executing on a Intel(R) Xeon(R) W-2133 CPU (3.60GHz) PC, with 32GB RAM running Windows 11 Enterprise. For expedience, we aborted any query using over 40 minutes of execution time.

Results: As we can see in Figure 4, the QURE translations outperform the ByePy ones for every single query, often by multiple orders of magnitude. QURE translations outperform the original UDFs for all queries except one (Q18[3]), by factors of up to 11x.

The main reasons for the observed improvement lie in QURE's approach being based on *verified lifting*, which seeks to identify *efficient* operators in the target language that map the source semantics. In contrast, ByePy relies heavily on the use of recursive CTEs in their translation, which allows it to translate more complex control flow structures than otherwise possible, but can result in highly inefficient SQL code. To illustrate this, consider the simplified version of our running example UDF below written in PL/pgSQL (used by [24, 25]):

```
CREATE  FUNCTION Test(key varchar) RETURNS varchar AS
DECLARE
  y varchar := ''; code varchar[];
BEGIN
  code := array['A','I'];
  FOR i IN 1..2 LOOP
    IF key = code[i] THEN
        y := code[i];
    END IF;
  END LOOP;
  RETURN y;
END;
```

After translation (using the online compiler at [5]), the resulting SQL UDF is 62 lines of code, with a 5 levels of nested `SELECT` statements (as opposed to the 1-liner generated by QURE in Figure 2), as depicted in Appendix B.

7.4 Comparison with PyFroid [21] and PyTond [57]

Our results on PyFroid and TPC-H Pandas workload suggest that our coverage is comparable. QURE is able to translate and verify complex expressions extracted from PyFroid workload and is able to verify equivalence of all 22 Pandas UDFs and its corresponding SparkSQL statements. However, the performance speed-ups reported in one system cannot be directly compared with those in another, especially since the scope of translations differs (e.g., a subset of expressions within a larger notebook vs. complete UDFs in Spark) which affects the runtimes.

[3]Here, the QURE translation performs an additional *unnest* operator not needed by the UDF, likely accounting for the increase in latency

7.5 LLM and Formal Verification Overheads

Typically, the compilation time for Spark queries with UDFs (without QURE) varies in a range of 0.5 to 2 seconds. In QURE, the overhead from LLM invocations averages around 4 seconds. Additionally, the formal verification process takes approximately 1 second on average, but can vary depending on the complexity and length of the translated Boogie code. On average, we find the total compilation overhead to be approximately 6 seconds.

For queries that process relatively small amounts of data per node, such as those involving Pandas scalar and aggregate UDFs over smaller datasets (e.g., SF = 10) and/or distributed across larger clusters (12 nodes), this overhead dominates runtime improvements. However, for analytic queries in big data systems, processing large amounts of data, the overhead from LLM inference and formal verification is minor compared to potential runtime improvements in orders of minutes.

Short-running queries, where the compilation overhead dominates, can potentially be identified using query performance prediction techniques like those described in [58, 60], or even simpler heuristics that consider estimated data processed per node or UDF (from query optimizer) to minimize performance regressions. For recurring queries, which are prevalent in big data systems [39], the historical performance data can be utilized to select queries that should be inlined, potentially performing this inlining offline. Determining for which queries QURE can have significant benefit up-front and with low latency is an interesting research challenge in its own right and out of scope for this paper. Finally, QURE can be used in offline query plan optimization or in a co-pilot setting, assisting users in inlining queries while they are composing or registering the UDFs.

7.6 Limitations of QURE and Future Work

There exist a number of limitations to the current version of QURE, besides the need for improvements to the number of Python constructs modeled.

(1) QURE currently does not try to repair errors in LLM translations; rather, it falls back to the original query, with a UDF, if verification fails. Reasoning about potential transformations for correcting SQL when verification fails may improve coverage. A related limitation not shared by competing techniques is the requirement to translate each UDF in its *entirety*, which may not be possible due to limitations of the target language.
(2) QURE is limited in the loop and control-flow complexity it can handle, as we highlight in Section 4.4; adding additional translation techniques and the corresponding Boogie modeling would increase QURE's reach.
(3) Adding pairs of equivalent functions to the function repository (see Section 3) is a manual process and requires the person matching the functions to understand the semantics and parameters of the functions added; semi-automated support and testing would be highly useful.
(4) While Apache Spark does not support it, several DBMSs allow SQL within UDFs; extending verification to these settings involving both SQL and procedural constructs is an interesting and challenging piece of future work.
(5) A further important area for future work is to provide a quick estimation of (a upper bound on) performance benefits from translation, allowing one to save translation time where not needed.

8 Related Work

Rule-based Translation (e.g., Froid [51], PyFroid [21], PyTond [57], ByePy [1]). Table 7 outlines the salient features of the different related techniques. While there are differences in the problem studied (code/script vs UDF to SQL translation) and different target system result in differences as to what can or cannot be supported, the primary difference lies in the focus of each technique. QURE's focus is on *verification of equivalence* between UDF code and SQL,

Table 7. Comparing QURE with relevant code to SQL translation approaches.

	QURE	PyFroid [21]/PyTond [57]	Froid [51]	ByePy [1]	QBS [17]
Problem	UDF to SQL (*Apache Spark*)	Script/code to SQL	UDF to SQL (*SQL Server*)	UDF to SQL (*PostgreSQL*)	Script/code to SQL (*Java*)
Core Focus	Equivalence verification between UDF and SQL	Synthesis of SQL from code	Synthesis of SQL from code	Synthesis of SQL from code	Synthesis of SQL-like expressions (focus) followed by verification
Approach for Inferring Verification Conditions	Leverages logical plan	No (NA)	No (NA)	No (NA)	Analysis of UDF procedural code
Targets Generic Python	Yes	No	No	Yes	No (NA)
Pandas DataFrames	Yes	Yes	No	No	No (NA)
UDFs w/ Embedded SQL	No (not supported in Spark)	No (NA)	Yes	Yes	No (NA)
Maps Multiple Expressions to a Single Equivalent (Intrinsic) SQL Operator/Function	Yes	Yes	Yes	No (procedural constructs translated to similar SQL expressions, and loops to recursive CTE)	Yes
Potential Performance Improvement	High	High	High	Improvement limited when compared with QURE, but translation extends to code with no direct SQL function/operator mapping	High

while, for the rest of the techniques, the primary focus is on *synthesis of SQL from UDF code*, which requires different innovations. Furthermore, QURE covers *both* Python and Pandas constructs unlike PyFroid, or PyTond. While ByePye covers such constructs (but is lacking support for DataFrames), we address a different problem of translating procedural constructs to (declarative) operations (also known as *verified lifting*) while ByePy translates UDF expressions to PostgreSQL expression without evaluating whether there is a single built-in SQL operator or function that is equivalent to a set of lines of UDF code. As we show in our ByePy experimental comparison, doing so almost always results in better performance. We believe the advantage of ByePy lies in cases when there is no direct mapping to SQL operations/functions, which will lead to non-translation with existing techniques such as QURE, PyFroid, or PyTond.

Program Synthesis and Formal Verification. In this line of work, QBS [17] is closely related to QURE, as it involves similar formal verification techniques. The primary difference lies in the synthesis of verification conditions: while QBS analyzes procedural code to synthesize complex logical assertions for invariants and post-conditions, QURE leverages the SQL plan, which reduces complexity by enabling much of the logical assertions to be pre-defined, requiring only lighter synthesis during online translation. QURE's verification techniques are particularly useful when AI or other tools can infer a candidate SQL query but require verification. Additional differences between QURE and QBS are highlighted in Table 7.

Other techniques use strategies such as counterexample-guided inductive synthesis (CEGIS) and incremental grammar generation to optimize UDFs within dataflow programs and ORM applications [18, 22, 30, 38, 52, 62]. Unlike QURE, these approaches do not guarantee equivalence of translated queries, necessitating human validation. For example, [61] employs bounded model checking (BMC), which explores programs for counterexamples up to a *certain depth, or bound*, and reports that the programs are equivalent within the explored bound. However, this method may miss discrepancies occurring at deeper states beyond the defined bound.

Compositional Program Synthesis. Compositional program synthesis has been explored to decompose the UDF-to-SQL translation problem [23, 48, 49], however, the synthesis challenges for complex expressions and diverse procedural constructs (e.g., different data structures, function calls, data frame handling) remain. In addition, most of these approaches used CEGIS-based translation approaches, thus requiring human validations.

DataFrame Parallelization and Translations. While orthogonal to UDF to SQL translation, techniques for improving scalability of Pandas DataFrame-based operations such as *Modin* [47], *dask* [53], or *PySpark* [8] have been proposed. However, each is limited to the specific subset of DataFrame operations parallelized by the framework. A few approaches, such as Grizzly [32] and

MagPie [40], support a rule-based translator that selectively translates specific DataFrame operations to SQL. However, they do not generalize well to complex expressions involving combinations of procedural constructs, e.g., loops, lists, dictionaries, and DataFrames.

SQL Query Optimization. Extensive research on optimizing SQL queries exists, especially those containing nested sub-queries [19, 20, 28, 29, 42, 44, 56]. While this is orthogonal to UDF to SQL translation, the translated SQL is often inlined as a subquery or CTEs, and some of these approaches can be used to optimize the query generated by QURE, which is an interesting area for future work.

9 Conclusion

QURE addresses the performance limitations of UDFs in Apache Spark by leveraging LLMs for the translation of Python/Pandas UDFs into equivalent Spark-SQL expressions and utilizing the semantics of operators in SQL plans for verification. By supporting a broader range of operations and procedural constructs, including third-party function calls and their compositions with SQL operations, QURE significantly extends the coverage beyond existing systems. Our empirical evaluation demonstrates that QURE not only achieves higher accuracy in translating and verifying UDFs but also delivers substantial performance improvements in SparkSQL queries, including reducing out-of-memory errors. Beyond these results, QURE's ability to leverage the logical plan allows extensibility, enabling it to easily support additional functions and operators beyond those currently integrated into the system.

Acknowledgments

We thank Nico Bruno, Badrish Chandramouli, Surajit Chaudhuri, Vivek Narasayya, Kaushik Rajan, and Karthik Ramachandra for the insightful discussions and feedback on this work. We also extend our gratitude to the anonymous reviewers at SIGMOD 2025 for their valuable feedback.

References

[1] Accessed 2025-01-10. Columnstore indexes guide. https://github.com/ByePy/examples.
[2] Accessed 2025-01-10. Generators. https://wiki.python.org/moin/Generators.
[3] Accessed: 2025-01-10. Logical and physical showplan operator reference. https://learn.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference?view=sql-server-ver16.
[4] Accessed: 2025-01-10. NumPy. https://numpy.org/.
[5] Accessed: 2025-01-10. Online UDF Compiler. https://apfel-db.cs.uni-tuebingen.de/.
[6] Accessed: 2025-01-10. Pandas Documentation. https://pandas.pydata.org/pandas-docs/stable/index.html.
[7] Accessed: 2025-01-10. Pandas.DataFrame.groupby. https://pandas.pydata.org/pandas/docs/stable/reference/api/pandas.DataFrame.groupby.html.
[8] Accessed: 2025-01-10. PySpark Overview. https://spark.apache.org/docs/latest/api/python/index.html.
[9] Accessed: 2025-01-10. QURE - MSR Webpage. https://www.microsoft.com/en-us/research/project/qure/.
[10] Accessed: 2025-01-10. Satisfiability Modulo Theories. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories.
[11] Accessed: 2025-01-10. Spark SQL, Built in Functions. https://spark.apache.org/docs/latest/api/sql/index.html.
[12] Accessed 2025-01-10. Uninterpreted Function. https://en.wikipedia.org/wiki/Uninterpreted_function.
[13] Accessed: 2025-01-10. What is a DataFrame? https://www.databricks.com/glossary/what-are-dataframes.
[14] Accessed: 2025-01-10. Z3 Prover. https://github.com/Z3Prover/z3.
[15] Andreas Blass and Yuri Gurevich. 2001. Inadequacy of Computable Loop Invariants. *ACM Trans. Comput. Logic* 2, 1 (2001), 1–11. https://doi.org/10.1145/371282.371285
[16] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Madan Musuvathi, Akash Lal, Aseem Rastogi, Nikhil Swamy, and Rahul Sharma. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *2023 Empirical Methods in Natural Language Processing*.
[17] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.
[18] Andrew Crotty et al. 2015. An architecture for compiling udf-centric workflows. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1466–1477.
[19] Umeshwar Dayal. 1987. Of Nests and Trees: A Unified approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the Very Large Data Bases (VLDB) Conference*.
[20] Manoj Elhemali, Cesar A. Galindo-Legaria, Torsten Grabs, and Milind M. Joshi. 2007. Execution Strategies for SQL Subqueries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
[21] K Venkatesh Emani, Avrilia Floratou, Carlo Curino, L Tanca, Q Luo, G Polese, L Caruccio, and X Oriol. 2024. PyFroid: Scaling Data Analysis on a Commodity Workstation.. In *EDBT*. 61–67.
[22] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for {Scale-Up} Architectures and {Medium-Size} Data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 799–815.
[23] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.
[24] Tim Fischer. 2023. To Iterate Is Human, to Recurse Is Divine – Mapping Iterative Python to Recursive SQL. In *BTW*. 1069–1074.
[25] Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *Proceedings of the International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 2389–2392.
[26] Kai Franz, Samuel Arch, Denis Hirn, Torsten Grust, Todd C. Mowry, and Andrew Pavlo. 2024. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *Conference on Innovative Data Systems Research*.
[27] Carlo Alberto Furia and Bertrand Meyer. 2010. *Inferring Loop Invariants Using Postconditions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 277–300.
[28] Cesar A. Galindo-Legaria and Milind Joshi. 2001. Orthogonal optimization of subqueries and aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 571–581.
[29] Robert A. Ganski and Harry K. T. Wong. 1987. Optimization of Nested SQL Queries Revisited. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
[30] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting Code Optimizations in {Data-Parallel} Pipelines through {PeriSCOPE}. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 121–133.
[31] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the curse of cursor loops using custom aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 559–573.

[32] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting pandas in a box. In *Conference on Innovative Data Systems Research (CIDR);(Online)*. 15.

[33] Denis Hirn. 2023. Data is Data and Control Should be Data, Too -— Compiling Iterative Table-valued PL/SQL UDFs into Recursive SQL Code. In *VLDB PhD Workshop*.

[34] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the International Conference on Management of Data*. Association for Computing Machinery.

[35] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *Proceedings of the International Conference on Management of Data*. Association for Computing Machinery, 723–735.

[36] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. doi:10.1145/363235.363259

[37] Catalin Hritcu. 2019. Evolution, Semantics, and Engineering of the F Verification System. (2019).

[38] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1292–1295.

[39] Alekh Jindal et al. 2018. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. 191–203.

[40] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends.. In *CIDR*.

[41] Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2024. Leveraging LLMs for Program Verification. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*. TU Wien Academic Press, 107–118.

[42] Won Kim. 1982. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems (TODS)* 7, 3 (1982).

[43] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[44] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. In *Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW) Conference*.

[45] Erik Nijkamp et al. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR* (2023).

[46] Erik Nijkamp et al. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *ICLR* (2023).

[47] Devin Petersohn et al. 2021. Flexible Rule-based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proc. VLDB Endow.* 15, 3 (2021), 739–751.

[48] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[49] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[50] Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proceedings of VLDB* 12, 12 (2019), 1810–1813. https://www.microsoft.com/en-us/research/publication/blackmagic-automatic-inlining-of-scalar-udfs-into-sql-queries-with-froid/

[51] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[52] Astrid Rheinländer, Martin Beckmann, Anja Kunkel, Arvid Heise, Thomas Stoltmann, and Ulf Leser. 2014. Versatile optimization of UDF-heavy data flows with sofa. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 685–688.

[53] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *14th Python in Science Conference*.

[54] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages *(NIPS '20)*. Red Hook, NY, USA.

[55] K. Rustan and M. Leino. Accessed: 2025-01-10. This is Boogie 2. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf.

[56] Prakash Seshadri, Hamid Pirahesh, and T. Y. Clarence Leung. 1996. Complex Query Decorrelation. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

[57] Hesam Shahrokhi et al. 2024. Pytond: Efficient python data science on the shoulders of databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 423–435.

[58] Shivaram Venkataraman et al. 2016. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.
[59] Wikipedia. Accessed: 2025-01-10. Abstract Syntax Tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree.
[60] Ziniu Wu et al. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data*. 280–294.
[61] Guoqiang Zhang, Benjamin Mariano, Xipeng Shen, and Işıl Dillig. 2023. Automated Translation of Functional Big Data Queries to SQL. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 580–608.
[62] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–26.

Appendix

(A) Examples of Few-Shot Prompt Templates for Translations

(A.1) Python UDFs

Task: Translate the following Python UDF to a Spark SQL expression. I will provide the UDF, its table source, attributes, and alias. Below are some examples.
// Example 1
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
Python UDF:

```
def udf(num):
    return num < 10
```

Equivalent SQL:

```
SELECT l_quantity < 10 AS udf_0_col
FROM lineitem
```

// Example 2
Table Source: `lineitem, orders`
Attributes: `l_shipmode, o_orderdate`
Alias: `udf_0_col`
Python UDF:

```
def udf(shipmode, orderdate):
    from date import fromisoformat
    if shipmode != "MAIL":
        return False
    if orderdate < date.fromisoformat("1994-01-01"):
        return False
    return True
```

Equivalent SQL:

```
SELECT (l_shipmode = 'MAIL' AND o_orderdate >= DATE '1994-01-01') AS
    udf_0_col
FROM lineitem, orders
```

// Example 3
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
Python UDF:

```
def not_less_than_10(num):
    if not num < 10:
        return True
```

```
    return False
```

Equivalent SQL:

```
SELECT l_quantity >= 10 AS udf_0_col
FROM lineitem
```

// UDF
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
Python UDF:

```
@udf(IntegerType())
def udf_0(num):
    if (num >= 0 and num < 10):
        return 0
    elif (num >= 10 and num < 20):
        return 1
    elif (num >= 20 and num < 30):
        return 2
    elif (num >= 30 and num < 40):
        return 3
    else:
        return 4
```

Equivalent SQL:

(A.2) Pandas Scalar UDFs

Task: Convert the following Pandas UDF to a Spark SQL expression. I will provide the UDF, its table source, attributes, alias, and group by columns if needed. Use `PARTITION BY` when necessary on provided group by columns. Below are some examples.
// Example 1
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
GroupBy: None
Python UDF:

```
@pandas_udf(DoubleType(), PandasUDFType.SCALAR)
def udf_0(df):
    return df + 1
```

Equivalent SQL:

```
SELECT l_quantity + 1 AS udf_0_col
FROM lineitem
```

// Example 2
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
GroupBy: `year(l_shipdate)`
Python UDF:

```
@pandas_udf(DoubleType(), PandasUDFType.SCALAR)
def udf_0(df):
    return df - df.mean()
```

Equivalent SQL:

```
SELECT l_quantity - mean(l_quantity)
OVER (PARTITION BY year(l_shipdate)) AS udf_0_col
FROM lineitem
```

// UDF
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
GroupBy: `year(l_shipdate), month(l_shipdate)`
Python UDF:

```
@pandas_udf(DoubleType(), PandasUDFType.SCALAR)
def udf_0(dfs):
    return (dfs - dfs.mean()) / dfs.std()
```

Equivalent SQL:

(A.3) Pandas Agg UDFs

Task: Convert the following Pandas aggregation UDF to a Spark SQL expression. Aggregate functions should use `GROUP BY`. Use `WHERE` instead of `HAVING`. Below are some examples.
// Example 1
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
Aggregate Attributes: `l_shipmode`
Python UDF:

```
@pandas_udf(IntegerType(), PandasUDFType.GROUPED_AGG)
def udf_0(df):
    return len(df.value_counts())
```

Equivalent SQL:

```
SELECT COUNT(DISTINCT l_quantity) AS udf_0_col
FROM lineitem
GROUP BY l_shipmode
```

// Example 2
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`
Aggregate Attributes: `l_shipmode`
Python UDF:

```
@pandas_udf(DoubleType(), PandasUDFType.GROUPED_AGG)
def udf_0(df):
    return df.mean()
```

Equivalent SQL:

```
SELECT AVG(l_quantity) AS udf_0_col
FROM lineitem
GROUP BY l_shipmode
```

// UDF
Table Source: `lineitem`
Attributes: `l_quantity`
Alias: `udf_0_col`

Aggregate Attributes: None
Python UDF:

```
@pandas_udf(DoubleType(), PandasUDFType.GROUPED_AGG)
def udf_0(df):
    return df.max() - df.min()
```

Equivalent SQL:

(A.4) Pandas Map UDFs

Task: Convert the following Pandas UDF to a Spark SQL expression. Aggregate functions should use `PARTITION BY` over aggregate attributes when necessary. Below are some examples.
// Example 1
Table Source: `lineitem`
Aggregate Attributes: `l_shipmode`
Python UDF:

```
@pandas_udf(SCHEMA, PandasUDFType.GROUPED_MAP)
def udf_0(df):
    l_quantity_col = df["l_quantity"]
    l_discount_mean = df["l_discount"].mean()
    return df.assign(l_quantity=l_quantity_col - l_discount_mean)
```

Equivalent SQL:

```
SELECT l_quantity - AVG(l_discount)
OVER (PARTITION BY l_shipmode) AS l_quantity
FROM lineitem
```

// Example 2
Table Source: `lineitem`
Aggregate Attributes: `l_shipmode`
Python UDF:

```
@pandas_udf(SCHEMA, PandasUDFType.GROUPED_MAP)
def udf_0(df):
    l_discount_col = df["l_discount"]
    return df.assign(l_discount=l_discount * 1000)
```

Equivalent SQL:

```
SELECT l_discount * 1000 AS l_discount
FROM lineitem
```

// UDF
Table Source: `lineitem`
Aggregate Attributes: None
Python UDF:

```
@pandas_udf(SCHEMA, PandasUDFType.GROUPED_MAP)
def udf_0(df):
    l_quantity_col = df["l_quantity"]
    return df.assign(l_quantity=l_quantity_col - 100)
```

Equivalent SQL:

(B) Translation from ByePy [5] for the example UDF in Section 7.3

```
CREATE OR REPLACE FUNCTION keytest_start(key varchar) RETURNS varchar
AS
$$
    WITH RECURSIVE run("rec?",
                       "label",
```

```
                        "res",
                        "key",
                        "y",
                        "code",
                        "i") AS
    (
        (SELECT True,
                'fori1_head',
                NULL :: varchar,
                "key",
                "y_1",
                "code_1",
                "i_1"
         FROM LATERAL (SELECT NULL :: varchar AS "y_4") AS "let11"("y_4"),
              LATERAL (SELECT NULL :: varchar[] AS "code_5") AS "let12"("code_5"),
              LATERAL (SELECT NULL :: int4 AS "i_6") AS "let13"("i_6"),
              LATERAL (SELECT '' AS "y_1") AS "let14"("y_1"),
              LATERAL
              (SELECT ARRAY['A','I'] :: text[] AS "code_1") AS "let15"("code_1"),
              LATERAL (SELECT 1 AS "i_1") AS "let16"("i_1"))
          UNION ALL
        (SELECT "result".*
         FROM run AS "run"("rec?", "label", "res", "key", "y", "code", "i"),
              LATERAL
              ((SELECT "ifresult2".*
                FROM LATERAL (SELECT 2 AS "q2_2") AS "let0"("q2_2"),
                     LATERAL (SELECT "i" <= "q2_2" AS "pred3_2") AS "let1"("pred3_2"),
                     LATERAL
                     ((SELECT "ifresult4".*
                       FROM LATERAL
                            (SELECT "key" = (("code")["i"]) AS "q7_3") AS "let3"("q7_3")
    ,
                            LATERAL
                            ((SELECT True, 'ifmerge6', NULL :: varchar, "key", "y_6", "
    code", "i"
                              FROM LATERAL (SELECT ("code")["i"] AS "y_6") AS "let5"("
    y_6")
                              WHERE NOT "q7_3" IS DISTINCT FROM True)
                               UNION ALL
                             (SELECT True, 'ifmerge6', NULL :: varchar, "key", "y", "
    code", "i"
                              WHERE "q7_3" IS DISTINCT FROM True)
                            ) AS "ifresult4"
                       WHERE NOT "pred3_2" IS DISTINCT FROM True)
                        UNION ALL
                      (SELECT False,
                              NULL :: text,
                              "y" AS "result",
                              "run"."key",
                              "run"."y",
                              "run"."code",
                              "run"."i"
                       WHERE "pred3_2" IS DISTINCT FROM True)
                     ) AS "ifresult2"
                WHERE "run"."label" = 'fori1_head')
                 UNION ALL
               (SELECT True, 'fori1_head', NULL :: varchar, "key", "y", "code", "i_5"
                FROM LATERAL (SELECT "i" + 1 AS "i_5") AS "let9"("i_5")
                WHERE "run"."label" = 'ifmerge6')
              ) AS "result"("rec?", "label", "res", "key", "y", "code", "i")
         WHERE "run"."rec?")
    )
    SELECT "run"."res" AS "res"
    FROM run AS "run"
    WHERE NOT "run"."rec?"
$$ LANGUAGE SQL;
```

(C) Boogie code for Nested Loop along with invariants

```
type KeyValue;

```

```
function getKey(kv: KeyValue): int;
function getValue(kv: KeyValue): int;
function joinPairs(kv1: KeyValue, kv2: KeyValue): KeyValue;
function keysMatch(elem1: KeyValue, elem2: KeyValue): bool;
function nestedLoopJoinOuterLoop_Props(data1: [int]KeyValue, len1: int, data2: [int]KeyValue, len2: int
      , result: [int]KeyValue, res_len: int, data1totallen: int, old_len: int): bool;
function nestedLoopJoinInnerLoop_Props(data1: [int]KeyValue, len1: int, data2: [int]KeyValue, len2: int
      , result: [int]KeyValue, res_len: int, data1totallen: int, data2totallen: int, old_len: int
      old_len: int): bool;

axiom (forall elem1: KeyValue, elem2: KeyValue :: (getKey(elem1) == getKey(elem2)) ==> keysMatch(elem1,
       elem2));
axiom (forall elem1: KeyValue, elem2: KeyValue :: (getKey(elem1) != getKey(elem2)) ==> !keysMatch(elem1
      , elem2));
axiom (forall data1: [int]KeyValue, len1: int, data2: [int]KeyValue, len2: int, result: [int]KeyValue,
      res_len: int, data1totallen: int, old_len: int ::
      (
       (len1 == 0)
       ||
       ((0 <= len1 && len1 <= len1 <= data1totallen)
       &&
       (old_len <= res_len && res_len < old_len + len2)
       &&
       !(exists x: int, y: int ::
         (0 <= x && x < len1) && (0 <= y && y < len2) &&
         keysMatch(data1[x], data2[y]) &&
         !existsInResult(data1[x], data2[y], result, 0, res_len)
        )
        &&
        !(exists x: int, y: int ::
         (0 <= x && x < len1) && (0 <= y && y < len2) &&
         !keysMatch(data1[x], data2[y]) &&
         existsInResult(data1[x], data2[y], result, 0, res_len)
        )
        &&
         (0 == res_len) ==>
             !(exists x: int, y: int ::
                 (0 <= x && x < len1) && (0 <= y && y < len2) &&
                 keysMatch(data1[x], data2[y])
         )
         &&
         ( (0 < res_len) ==>
             ( forall z: int :: (0 <= z && z < res_len) ==>
             (exists x: int, y: int ::
                (0 <= x && x < len1) && (0 <= y && y < len2) &&
                 keysMatch(data1[x], data2[y]) && result[z] == joinPairs(data1[x], data2[y])
             )
            )
          )
     )
     ==> nestedLoopJoinOuterLoop_Props(data1, len1, data2, len2, result, res_len) == true
);


axiom (forall data1: [int]KeyValue, len1: int, data2: [int]KeyValue, len2: int, result: [int]KeyValue,
      res_len: int, data1totallen: int, data2totallen: int, old_len: int ::
      (
        ((len2 == 0)
         ||
   ((0 <= len2 && len2 <= data2totallen)
        &&
   (old_len <= res_len && res_len < old_len + 1)
      &&
      !(exists x: int, y: int ::
        (x == len1) && (0 <= y && y < len2) &&
        keysMatch(data1[x], data2[y]) &&
        !existsInResult(data1[x], data2[y], result, old_len, res_len)
       )
       &&
       !(exists x: int, y: int ::
        (x == len1) && (0 <= y && y < len2) &&
        !keysMatch(data1[x], data2[y]) &&
        existsInResult(data1[x], data2[y], result, old_len, res_len)
       )
       &&
        (old_len == res_len) ==>
            !(exists x: int, y: int ::
                (x == len1) && (0 <= y && y < len2) &&
                keysMatch(data1[x], data2[y])

        )
```

```
        &&
        ( (old_len < res_len) ==>
            ( forall z: int :: (old_len <= z && z < res_len) ==>
            (exists x: int, y: int ::
                (x == len1) && (0 <= y && y < len2) &&
                keysMatch(data1[x], data2[y]) && result[z] == joinPairs(data1[x], data2[y])
            )
           )
         )
    )
    ==> nestedLoopJoinInnerLoop_Props(data1, len1, data2, len2, result, res_len, old_len) == true
);


function existsInResult(data1_element: KeyValue, data2_element: KeyValue, result: [int]KeyValue,
     old_len: int, res_len: int) : bool;

axiom (forall data1_element: KeyValue, data2_element: KeyValue, result: [int]KeyValue, old_len: int,
     res_len: int ::
    (exists z: int :: (old_len <= z && z < res_len) &&
     (result[z] == joinPairs(data1_element, data2_element))
    ) ==> existsInResult(data1_element, data2_element, result, old_len, res_len) == true
);

axiom (forall data1_element: KeyValue, data2_element: KeyValue, result: [int]KeyValue, old_len: int,
     res_len: int ::
    !(exists z: int :: (old_len <= z && z < res_len) &&
     (result[z] == joinPairs(data1_element, data2_element))
    ) ==> existsInResult(data1_element, data2_element, result, old_len, res_len) == false
);

axiom (forall data1_element: KeyValue, data2_element: KeyValue, result: [int]KeyValue, old_len: int,
     res_len: int ::
    (old_len == res_len) ==> existsInResult(data1_element, data2_element, result, old_len, res_len) ==
     false
);

function countJoinPairsRecursive(data1: [int]KeyValue, i: int, data2: [int]KeyValue, len2: int): int {
    if (i == 0) then 0
    else countJoinPairsRecursive(data1, i - 1, data2, len2) + countInnerPairs(data1[i - 1], data2, len2
     , len2)
}

function countInnerPairs(element: KeyValue, data2: [int]KeyValue, j: int, len2: int): int {
    if (j == 0) then 0
    else countInnerPairs(element, data2, j - 1, len2) + (if (getKey(element) == getKey(data2[j - 1]))
     then 1 else 0)
}


procedure nestedloopsjoin(data1: [int]KeyValue, len1: int, data2: [int]KeyValue, len2: int)
    returns (result: [int]KeyValue, res_len: int)
requires len1 > 0;
requires len2 > 0;
{
    var i, j: int;
    var old_len: int;

    res_len := 0;
    i := 0;
     while (i < len1)
     invariant nestedLoopJoinOuterLoop_Props(data1, i, data2, len2, result, res_len, len1, old_len);
     {
        j := 0;
        old_len := res_len;
        while (j < len2)
        invariant nestedLoopJoinInnerLoop_Props(data1, i, data2, j, result, res_len,  len1, len2,
     old_len);
        {
            if (getKey(data1[i]) == getKey(data2[j]))
            {
                result[res_len] := joinPairs(data1[i], data2[j]);
                res_len := res_len + 1;
            }
            j := j + 1;
        }
        i := i + 1;
    }
}
```

(D) Boogie code for Group-By Sum Aggregation

```
type KeyValue;
function getKey(kv: KeyValue): int;
function getValue(kv: KeyValue): int;
function lookup(ad: [int]int, k: int, len: int) returns (int);
function isDistinctKeyArray(arr: [int]int, len: int): bool;
function containsAllDistinctKeys(data: [int]KeyValue, data_len: int, aggregated_data: [int]int, ad_len: int): bool;
function sumValuesForKeyData(data: [int]KeyValue, keys_array: [int]int, ad_len: int, p: int, len: int): int;
function Groupbysum_Props(data: [int]KeyValue, keys_array: [int]int, values_array: [int]int, ad_len: int,  data_len: int, cur_iter: int): bool

axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
       (0 <= lookup(ad, k, len) && lookup(ad, k, len) < len) ==> (getKey(data[lookup(ad, k, len)]) == k));

axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
       (lookup(ad, k, len) == len) <==> (forall i: int :: (0 <= i && i < len) ==> (getKey(data[i]) != k)));

axiom (forall ad: [int]int, data: [int]KeyValue, k: int, len: int ::
        len == 0 ==> (lookup(ad, k, len) == -1));

axiom (forall ad: [int]int, k1: int, k2: int, len: int ::
       (0 <= lookup(ad, k1, len) && 0 <= lookup(ad, k2, len) && k1 != k2) ==> (lookup(ad, k1, len) != lookup(ad, k2, len)));

axiom (forall ad: [int]int, k1: int, k2: int, len: int, len2: int ::
       (0 <= lookup(ad, k1, len) && 0 <= lookup(ad, k2, len2) && len <= len2) ==> (lookup(ad, k1, len) != lookup(ad, k2, len2)));

axiom (forall ad: [int]int, k: int, len: int, len2: int ::
       (0 <= lookup(ad, k, len) && 0 <= lookup(ad, k, len2) && len <= len2) ==> (lookup(ad, k, len) == lookup(ad, k, len2)));



axiom (forall arr: [int]int, len: int ::
       (forall i: int :: (0 <= i && i < len) ==> (lookup(arr, arr[i], i) == -1))
       &&
       (forall i, j: int :: (0 <= i && i < j && j < len) ==> (arr[i] != arr[j]))
  => isDistinctKeyArray(arr, len)
       );


axiom (forall data: [int]KeyValue, data_len: int, aggregated_data: [int]int, ad_len: int ::
       (forall i: int :: (0 <= i && i < data_len) ==>
           (lookup(aggregated_data, getKey(data[i]), ad_len) != -1))
      &&
      (forall j: int :: (0 <= j && j < ad_len) ==> (exists k: int :: (0 <= k && k < data_len && getKey(data[k]) == aggregated_data[j])
     ==> containsAllDistinctKeys(data, data_len, aggregated_data, ad_len)
      );


axiom (forall data: [int]KeyValue, keys_array: [int]int, ad_len: int, data_len: int, cur_iter: int ::
((cur_iter == 0)
  ||
  ((0 <= cur_iter && cur_iter <= data_len)
  &&
        (0 <= ad_len && ad_len <= cur_iter)
  &&
  containsAllDistinctKeys(data, data_len, keys_array, ad_len)
  &&
        isDistinctKeyArray(keys_array, ad_len)
  &&
        (forall p: int :: 0 <= p && p < ad_len && values_array[p] == sumValuesForKeyData(data,keys_array, ad_len, keys_array[p], ad_ cur_iter)))
       ==> Groupbysum_Props(data, keys_array, values_array, ad_len,  data_len, cur_iter) == True;

axiom (forall data: [int]KeyValue, keys_array: [int]int, ad_len: int, p: int, len: int ::
       (len == 0) ==> (sumValuesForKeyData(data, keys_array, ad_len, p, len) == 0));


axiom (forall data: [int]KeyValue, keys_array: [int]int, ad_len: int, p: int, len: int ::
       (len > 0) ==> (sumValuesForKeyData(data, keys_array, ad_len, p, len) ==
                      sumValuesForKeyData(data, keys_array, ad_len, p, len - 1) +
                      (if lookup(keys_array, getKey(data[len - 1]), ad_len) == p then getValue(data[len - 1]) else 0)));
```

```
69
70
71 procedure udf(data: [int]KeyValue, data_len: int) returns (keys_array: [int]int, values_array: [int]int
       , ad_len: int)
72 requires data_len > 0;
73 {
74     var i, j, k, v: int;
75     ad_len := 0;
76     i := 0;
77
78     while (i < data_len)
79    invariant Groupbysum_Props(data, keys_array, values_array, ad_len,  data_len, i)
80     {
81         k := getKey(data[i]);
82         v := getValue(data[i]);
83         j := lookup(keys_array, k, ad_len);
84
85         if (j == ad_len){
86           keys_array[ad_len] := k;
87           values_array[ad_len] := v;
88           ad_len := ad_len + 1;
89         }
90         else{
91          values_array[j] := values_array[j] + v;
92         }
93         i := i + 1;
94     }
95 }
```