Optimal Predicate Pushdown Synthesis

ROBERT ZHANG, University of Texas at Austin, USA
ERIC HAYDEN CAMPBELL, University of Texas at Austin, USA
DIXIN TANG, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA

Predicate pushdown is a long-standing performance optimization that filters data as early as possible in a computational workflow. In modern data pipelines, this transformation is especially important because much of the computation occurs inside *user-defined functions (UDFs)* written in general-purpose languages such as Python and Scala. These UDFs capture rich domain logic and complex aggregations and are among the most expensive operations in a pipeline. Moving filters ahead of such UDFs can yield substantial performance gains, but doing so requires *semantic* reasoning. This paper introduces a general semantic foundation for predicate pushdown over stateful fold-based computations.

We view pushdown as a correspondence between two programs that process different subsets of input data, with correctness witnessed by a *bisimulation invariant* relating their internal states. Building on this foundation, we develop a sound and relatively complete framework for verification, alongside a synthesis algorithm that automatically constructs *optimal pushdown decompositions* by finding the strongest admissible pre-filters and weakest residual post-filters. We implement this approach in a tool called Pusharoo and evaluate it on 150 real-world pandas and Spark data-processing pipelines. Our evaluation shows that Pusharoo is significantly more expressive than prior work, producing optimal pushdown transformations with a median synthesis time of 1.6 seconds per benchmark. Furthermore, our experiments demonstrate that the discovered pushdown optimizations speed up end-to-end execution by an average of 2.4× and up to two orders of magnitude.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Theory of computation** → **Program verification**; *Logic and verification*; *Automated reasoning*; *Invariants.*

Additional Key Words and Phrases: predicate pushdown, bisimulation, user-defined functions, data pipelines

ACM Reference Format:
Robert Zhang, Eric Hayden Campbell, Dixin Tang, and Işıl Dillig. 2026. Optimal Predicate Pushdown Synthesis. *Proc. ACM Program. Lang.* 10, PLDI, Article 234 (June 2026), 25 pages. https://doi.org/10.1145/3808312

1 Introduction

Program optimizations that reorder computation are pervasive in modern data-processing systems. One of the most fundamental among them is *predicate pushdown*, which seeks to move filtering conditions earlier in a data pipeline so that irrelevant data is discarded before expensive operations are applied [18 (Ch. 16.2.3), 44]. As illustrated in Figure 1, this transformation replaces a pipeline that filters results *after* a computation with an equivalent one that filters inputs (partially or fully) *before* the computation, thereby reducing downstream computation cost.

Authors' Contact Information: Robert Zhang, University of Texas at Austin, Austin, USA, robertz@cs.utexas.edu; Eric Hayden Campbell, University of Texas at Austin, Austin, USA, eric.hayden.campbell@utexas.edu; Dixin Tang, University of Texas at Austin, Austin, USA, dixin@utexas.edu; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART234
https://doi.org/10.1145/3808312

This optimization is particularly important in modern data pipelines, where much of the computation occurs inside *user-defined functions (UDFs)* written in general-purpose programming languages such as Python and Scala. These functions let developers express rich domain logic and complex aggregations, but they are also among the *most expensive* operations in a pipeline. Moving filters ahead of such computations can therefore yield substantial performance gains by reducing the amount of data processed by a UDF. The challenge, however, is that many UDFs are fold-like computations that maintain complex internal state and involve nested control flow, requiring rich semantic reasoning for correctness. Recent work [48] has begun to explore predicate pushdown for restricted classes of UDFs, but a robust, general account of predicate pushdown for complex, stateful UDFs has yet to be systematically studied.

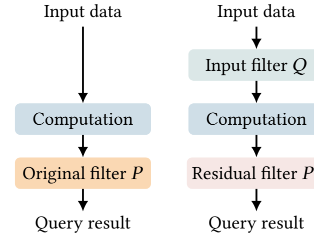

Fig. 1. Original (left) and optimized (right) pipelines

This paper aims to fill exactly this gap. As a starting point, we develop a unified semantic foundation of predicate pushdown for stateful UDFs, which we model as pure, deterministic folds whose observable behavior is captured by the evolution of an internal accumulator state. Within this framework, a pushdown transformation is interpreted as a semantic correspondence between two state-transforming programs that operate on different subsets of input data. Correctness then amounts to proving a relational property, namely, a *bisimulation invariant* [34], that links the executions of the original and transformed programs. This formulation accommodates the full generality of stateful UDFs, whose internal states may evolve differently before and after pushdown, but must converge to the same final result. From this foundation, we then develop a sound and relatively complete verification framework for candidate pushdown optimizations.

Building on this verification framework, we then introduce a new synthesis algorithm for automatically discovering *optimal pushdown decompositions* of a post-computation filter P. Specifically, the algorithm splits P into a pre-computation filter Q that is as logically strong as possible, and a *residual* filter P' that is as weak as possible, within a fixed predicate universe. This notion of optimality is important because it maximizes early pruning of input data while also minimizing redundant post-UDF validation. However, a key challenge in solving this optimal synthesis problem is that the algorithm must jointly infer three tightly-coupled proof artifacts: a (strongest) input-side filter Q, a (weakest) residual filter P', and a bisimulation invariant that witnesses their correctness. Even ignoring the optimality requirement, this problem lies beyond the reach of existing methods. For instance, unlike many predicate synthesis tasks that reduce to solving Constrained Horn Clauses (CHCs), the verification conditions connecting these unknown predicates are inherently *non-Horn*, making them incompatible with off-the-shelf CHC solvers. On top of this, the optimality requirement further complicates the search, as Q and P' must be optimized in opposite directions.

Our synthesis method solves these challenges through three complementary algorithmic innovations. First, we structurally decompose the search space into tractable sub-problems *without* compromising optimality. Second, we obtain sound under- and over-approximations for any valid bisimulation invariant and use these symbolic bounds as *unrealizability certificates* during synthesis. Finally, we incorporate domain-specific *root cause analysis* for diagnosing failed synthesis attempts and use this to inform *predicate repair*. Collectively, these techniques effectively navigate the search space in a principled way, ultimately guaranteeing both correctness and optimality.

We implemented our approach in a tool called Pusharoo and evaluated it on 150 real-world benchmarks drawn from pandas and Apache Spark workloads. Pusharoo successfully synthesizes correct pushdown transformations for all benchmarks and substantially outperforms prior work, which can handle less than 15% and often yields suboptimal decompositions. Furthermore, the algorithm runs efficiently in practice, with a median synthesis time of 1.6 seconds. Finally, the

optimizations enabled by our method result in significant speed-ups of real-world data analytics pipelines, yielding an average speed-up of 2.4× and up to two orders of magnitude in some cases.

To summarize, this paper makes the following key contributions:

- We introduce a unified semantic formulation of predicate pushdown that subsumes and extends prior definitions.
- We present a sound and relatively complete methodology for verifying the correctness of candidate pushdown transformations for stateful computations.
- We develop a novel synthesis algorithm that can generate *optimal* and *provably correct* pushdown optimizations. Notably, our method simultaneously searches for an optimal pushdown optimization together with its certificate of correctness.
- We empirically evaluate our method on 150 real-world benchmarks and demonstrate its effectiveness in terms of (1) expressiveness, (2) optimization time, and (3) the end-to-end performance improvements it enables.

2 Background

In this section, we review the two main forms of predicate pushdown studied in prior work and introduce the class of user-defined functions (UDFs) to which our formulation applies. Since UDFs serve as the building blocks of modern data analytics, we first discuss how they are structured and used in practice. Throughout the paper, we use the terms *UDF* and *computation* interchangeably.

2.1 User-Defined Functions in Data Pipelines

In this paper, we consider a general class of UDFs that can be expressed as as *state-transforming folds*, i.e., computations that traverse a sequence of input rows while updating an internal state. Each UDF operates over a *dataframe*, which we model as a finite sequences of tuples, sometimes referred to as *rows*. We assume that each UDF can be expressed in terms of a left fold:

$$F(x) = \mathsf{fold}(x, I, f) = f(\dots f(f(I, r_1), r_2), \dots, r_n) \tag{1}$$

where $x = [r_1, \dots, r_n]$ is the input dataframe, I is the initial state, and $f : A \times R \to A$ is a pure, deterministic update function called the *accumulator*. The *internal state* of type A is initialized to I and updated row by row by applying f. We adopt the convention that $\mathsf{fold}([], I, f)$ yields a distinguished undefined value $\perp$, and we assume that UDF evaluation is pure with respect to the pipeline. That is, external side effects, such as I/O and mutation of global state, are out of scope unless they are explicitly reified in the internal state. However, we do not impose restrictions on the structure of the internal state, which may be a scalar, tuple, list, or even a dataframe. Thus, our formalism can model a wide variety of UDFs, including:

- **Row-wise UDFs** correspond to folds where the input dataframe consists of a single row, and the accumulator f applies a stateless transformation to that row.
- **Aggregation UDFs** summarize the entire input into a single output value. These functions typically maintain a running summary (e.g., sum, maximum, or top-k elements), which f updates incrementally across the sequence of rows.
- **Table-valued UDFs** produce a variable number of output rows. In our model, these are captured by using a list- or dataframe-valued internal state, where f appends new rows.

While this fold-based view captures the core behavior of individual UDFs, such functions rarely appear in isolation. In practice, they are embedded within larger data-processing workflows that combine multiple relational operators (such as joins, group-bys, and filters) with user-defined computations. We refer to such workflows as *pipelines* and provide two simple examples below.

```python
# Filter to premium items
premium = df[df['category'] == 'premium']
# Compute discounted prices for selected items
discounted = premium.apply(
  lambda r: r['price'] * 0.9, axis=1)
# Retain those w/ discounted price at/above $900
filtered = discounted[discounted >= 900]
```

(a) Original pipeline

```python
# Filter to premium items at/above $1000
prefiltered = df[
  (df['category'] == 'premium') &
  (df['price'] >= 1000)]
# Apply discounting UDF after filtering
discounted = prefiltered.apply(
  lambda r: r['price'] * 0.9, axis=1)
```

(b) Exact pushdown

Fig. 2. (a) A pipeline that retrieves premium items with a discounted price at or above $900 (b) Exact pushdown: the post-processing filter is pushed through the discounting UDF entirely.

Example 2.1. Figure 2a shows a pandas pipeline that (1) selects only `premium` items based on the `category` attribute, (2) applies a row-wise UDF that computes a 10% discount by multiplying the `price` field by 0.9, and (3) retains only those items whose discounted price is at least $900.

Example 2.2. Figures 3a and 3b show a pipeline involving a user-defined aggregation. Unlike the previous row-wise UDF, `top2` maintains state across rows to compute the top two values in a group. The pipeline in Figure 3b applies this UDF per `team`, extracting the two highest `scores` for each team, then applies a filter to keep only teams whose top scores both exceed 90.0.

2.2 Two Flavors of Predicate Pushdown Optimizations

In this section, we describe two different types of pushdown optimizations considered in prior work [33, 44, 48], namely, *exact* (*equivalent*) pushdown and *partial* (*superset*) pushdown. Throughout this section, we use the letter P to denote a predicate over the output of the UDF and Q to denote a predicate over *an input row*.

Exact Pushdown. Given a computation F followed by a predicate P, an *exact (equivalent) pushdown* optimization can be applied if the post-computation predicate P can be entirely eliminated. Formally, exact pushdown seeks a *pre-computation* predicate Q satisfying the following condition:

$$\forall x.\ \big(y = F(x) \land z = F(\text{filter}_Q(x))\big) \implies \big((P(y) \Rightarrow y = z)\ \land\ (\neg P(y) \Rightarrow z = \bot)\big),$$

where $\text{filter}_Q(x)$ retains only elements of x satisfying Q, and $\bot$ denotes the absence of any result, treated as failing any filter (i.e., $P(\bot)$ is false). Thus Q is a valid pushdown iff applying it before F yields the same outcome as applying P after F, the classical form in relational query optimization.

Example 2.3. Recall the pipeline from Figure 2a, which computes discounted prices using a row-wise UDF and then filters the results to retain only items priced at or above $900. Let F denote the discounting UDF and $P(a) = (a \geq 900)$ be the post-UDF filter. Since $F(r) = r[\text{`price'}] \cdot 0.9$, the condition $P(F(r))$ holds iff $r[\text{`price'}] \geq 1000$. Thus, we can push the predicate through F exactly by defining $Q(r) = (r[\text{`price'}] \geq 1000)$, as shown in Figure 2b. This transformation satisfies the definition of exact pushdown: it guarantees that the output is identical to the original, since rows that would have failed P are excluded and those that would have passed are included.

Partial Pushdown. When exact pushdown is not feasible, prior work [33, 48] has also considered a related optimization called *partial (superset) pushdown*. This variant performs filtering before executing F, but also applies the original filter P afterwards. In other words, it may conservatively preserve additional data that would cause P to fail, and then filters those "false positives" out using P after F is applied. Formally, partial pushdown is possible if there exists a predicate Q such that:

$$\forall x.\ \big(y = F(x) \land z = F(\text{filter}_Q(x))\big) \implies \big((P(y) \Rightarrow P(z))\ \land\ (P(z) \Rightarrow y = z)\big)$$

```python
# User-defined aggregation function
def top2(x):
  fst = snd = float('-inf')
  for r in x:
    if r > fst: snd, fst = fst, r
    elif r > snd: snd = r
  return [fst, snd]
```

(a) UDF top2

```python
top2s = (df
  .groupby('team')['score']
  .agg(top2))
# P: keep only teams whose two highest scores
#    both exceed 90.0
filtered = top2s[top2s.apply(lambda a:
  a[0] > 90.0 and a[1] > 90.0)]
```

(b) Original pipeline

```python
# Filter out low scores before aggregation
prefiltered = df[df['score'] > 90.0]
top2s = (prefiltered
  .groupby('team')['score']
  .agg(top2))
# Apply full predicate after aggregation
filtered = top2s[top2s.apply(lambda a:
  a[0] > 90.0 and a[1] > 90.0)]
```

(c) Partial pushdown

```python
prefiltered = df[df['score'] > 90.0] # Q
top2s = (prefiltered
  .groupby('team')['score']
  .agg(top2))
# P': need only check whether a team has seen
#     any second-highest score
filtered = top2s[top2s.apply(lambda a:
  a[1] != float('-inf'))]
```

(d) Split pushdown

Fig. 3. (a) A UDF that computes the two highest scores for each team. (b) A pipeline using the UDF to retrieve teams whose two highest scores exceed 90.0. (c) Partial pushdown: the input filter eliminates rows that cannot help satisfy the post-computation filter. (d) Split pushdown: the post-processing predicate is weakened to P'.

Intuitively, Q is a conservative over-approximation of P: the first clause $P(y) \implies P(z)$ ensures *soundness* (no valid result is lost), and the second $P(z) \implies y = z$ ensures *consistency* (any accepted output matches the original).

Example 2.4. Consider again the pipeline from Figures 3a and 3b, which uses a UDF to compute the top two scores for each team. Let F denote the top2 function and $P(a) = (a[0] > 90.0 \wedge a[1] > 90.0)$ be the post-UDF filter from Figure 3b. This predicate requires that both the highest and second-highest scores of a team exceed the threshold. For this example, exact pushdown is not possible because any row-wise test $Q(r)$ can only inspect one score at a time and *cannot* decide whether a given row will end up as the highest or second highest without having access to the entire dataframe. However, *partial* pushdown is possible using the predicate $Q(r) = (r[\text{`score'}] > 90.0)$, which eliminates only those rows that cannot contribute to satisfying the post-filter P. Since a team may not satisfy P after aggregation (e.g., if only one score exceeds the threshold), the partially pushed-down pipeline re-applies P, as shown in Figure 3c.

3 Problem Statement

We now introduce a unified formulation of predicate pushdown that generalizes both exact and partial pushdown. This formulation enables a single algorithm that can synthesize a wide range of pushdown strategies, including those that fall strictly outside the exact and partial cases.

First, given a predicate P and element a, we define a function $\text{Lift}(P, a)$ that returns the result of the computation only when P is satisfied and yields $\perp$ otherwise:

$$\text{Lift}(P, a) = a \text{ if } P(a), \text{ else } \perp$$

Intuitively, Lift decides whether to keep or discard an internal state based on P, with $\perp$ playing the role for internal states that the empty sequence plays for dataframes. We can now define a unified formulation of predicate pushdown as follows:

Definition 3.1. **(Generalized Predicate Pushdown)** Let $F : [R] \rightarrow A$ be a computation and $P : A \rightarrow \text{Bool}$ a predicate on its output. Generalized predicate pushdown seeks to find a pair of predicates $Q : R \rightarrow \text{Bool}$ and $P' : A \rightarrow \text{Bool}$ such that the following equivalence holds:

$$\forall x. \quad \mathsf{Lift}(P, F(x)) \;=\; \mathsf{Lift}(P', F(\mathsf{filter}_Q(x))) \tag{2}$$

Intuitively, the input-side filter Q *pre-filters* the input to F, removing rows that can never contribute to satisfying P, and P' is a *residual* predicate over F's output that may be strictly weaker than P and captures *exactly* the information not preserved by Q. We refer to any pair (Q, P') satisfying Equation (2) as a *solution* to the *generalized* predicate pushdown problem. As the following theorem shows, this formulation subsumes both exact and partial pushdown as special cases, recovering them by choosing P' to be true or P, respectively.[1]

Proposition 3.2. *Let F be a computation on inputs x and P a predicate. For any input-side filter Q:*
(1) Q satisfies the exact pushdown *condition for (F, P) precisely when $(Q, true)$ satisfies Equation (2)*
(2) Q satisfies the partial pushdown *condition for (F, P) precisely when (Q, P) satisfies Equation (2).*

Example 3.3. The pipeline from Figure 3d illustrates a case where our formulation admits a decomposition that is different from both exact and partial pushdown. Recall that the post-UDF filter P in this example is $a[0] > 90.0 \wedge a[1] > 90.0$ and that *partial* pushdown can be applied with $Q(r) = (r[\text{'score'}] > 90.0)$. Using the same Q, our formulation enables a strictly weaker residual predicate, namely $P'(a) = (a[1] \neq -\infty)$, which removes redundant checks. In particular, the original predicate P explicitly verifies both scores against the 90.0 threshold. However, after applying Q, any defined top scores must already be above this threshold. Thus, the check performed by P after filtering is partially redundant, and the weaker predicate P' avoids this redundancy.

As illustrated in this example, some pipelines admit a decomposition in which the input filter Q rules out some, but not all, irrelevant rows, and the residual predicate P' still enforces part of the original post-computation filter. We refer to such cases as instances of **split pushdown**, reflecting that responsibility for enforcing the filter is divided between Q and P'.

Optimal Decomposition. While Equation (2) admits sound decompositions that go beyond exact and partial pushdown, it also admits trivial solutions: for instance, taking $Q \equiv \text{true}$ and $P' \equiv P$ always satisfies Equation (2). To avoid such vacuous solutions and realize the practical benefits of predicate pushdown, we additionally seek decompositions that maximize input reduction and minimize post-processing effort. Intuitively, we want the input filter Q to be as *strong* as possible so that irrelevant rows are excluded early, but we want the residual P' to be as *weak* as possible so that the post-UDF check is minimal. We formalize these goals as follows:

Definition 3.4. **(Optimal Predicate Pushdown)** Let $F : [R] \rightarrow A$ be a computation and $P : A \rightarrow \text{Bool}$ be a predicate on its output. An optimal solution to the generalized predicate pushdown problem is a pair (Q^*, P^*), where $Q^* : R \rightarrow \text{Bool}$ and $P^* : A \rightarrow \text{Bool}$, such that

$$\forall x. \quad \mathsf{Lift}(P, F(x)) = \mathsf{Lift}(P^*, F(\mathsf{filter}_{Q^*}(x))) \tag{3}$$

and (Q^*, P^*) satisfies the following optimality conditions:

- **Strongest input filter:**Among all solutions to Equation (3), Q^* is maximal under implication. That is, there is no other pair (Q', P') satisfying Equation (3) such that $Q' \Rightarrow Q^*$ and $Q^* \not\Rightarrow Q'$.
- **Weakest residual check:** Fixing the input filter Q^*, among all solutions to Equation (3) with Q^*, the residual P^* is minimal under implication. That is, there is no other pair (Q^*, P') satisfying Equation (3) such that $P^* \Rightarrow P'$ and $P' \not\Rightarrow P^*$.

[1]Proofs of all theorems are provided in Appendix ??.

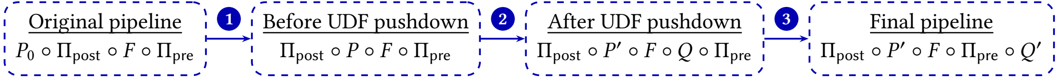

Fig. 4. Illustration of how our problem formulation fits into end-to-end pipeline optimization. Steps 1 and 3 can be performed by existing relational optimizers, whereas step 2 is enabled by our formulation.

Remark 1: Existence of optimal solutions. In general, an optimal decomposition need not exist as a finitely representable predicate: one can have a strictly improving sequence of feasible decompositions with no strongest input filter (or weakest residual) expressible in the language. Accordingly, throughout the paper, optimality is understood relative to fixed finite spaces of candidate predicates, within which an optimal decomposition always exists.

Remark 2: Impact on performance. While optimality is defined in terms of logical strength and weakness, these properties have clear practical consequences in terms of performance. A stronger pre-computation filter Q excludes more rows upfront, directly reducing UDF work. Minimizing the residual predicate is more nuanced: logical weakness does not, by itself, imply cheaper evaluation. In our setting, however, we restrict P' to mostly be a conjunction of a subset of the conjuncts in P, so minimizing P' mainly amounts to dropping conjuncts already enforced by Q and thus avoiding redundant post-computation checks.

Remark 3: Using UDF pushdown for pipeline optimization. UDFs typically sit inside larger pipelines of joins, projections, and filters. Figure 4 shows how our generalized pushdown formulation composes with standard relational rewrites: starting from $P_0 \circ \Pi_{\text{post}} \circ F \circ \Pi_{\text{pre}}$, a host optimizer first pushes P_0 left through Π_{post} using UDF-agnostic rules [44], yielding a predicate P at the UDF boundary ("Before UDF pushdown"). Our formulation then rewrites $P \circ F$ as $P' \circ F \circ Q$ with (Q, P') satisfying Equation (2) ("After UDF pushdown"). The UDF no longer blocks subsequent rewrites: Q and any additional predicates can then be pushed right through Π_{pre} as Q' ("Final pipeline").

4 Verification Methodology

In this section, we present a verification method for checking the correctness of a candidate pushdown optimization. The central challenge in verifying pushdown transformations is that the optimized and original executions no longer process the same sequence of inputs: the pushed-down version skips rows filtered by Q, while the original processes them all. As a result, their internal states may differ during execution. To reason about such executions, we introduce a *bisimulation invariant* $\psi(a_1, a_2)$ that relates the internal state a_1 of the original computation to the internal state a_2 of the optimized one. Intuitively, ψ captures a relational correspondence between these two internal states: it allows temporary differences caused by skipped rows, but requires that the executions eventually agree on the same final output.

Figure 5 formalizes this reasoning principle through four verification conditions (VCs) that together ensure the soundness of a pushdown transformation: *initialization*, *synchronized-step preservation*, *stutter-step preservation*, and *final-state agreement*. These conditions correspond to the base case, inductive steps, and postcondition of an inductive proof, and they enable local reasoning about symbolic accumulator states without explicitly tracking full input traces. We explain these VCs in more detail below.

Initialization (Init). The first VC asserts that ψ holds at the beginning of both executions, when the state is initialized to I. This serves as the base case for the inductive proof.

Synchronized-step preservation (Sync). This condition handles the case where both the original and optimized executions process the same input row r (i.e., when $Q(r)$ holds). In this case, both

Verification Conditions for Generalized Pushdown. Let ψ denote the bisimulation invariant relating the internal states of the original and optimized executions. The pushdown pair (Q, P') is sound if the following hold for all $a_1, a_2 \in A$ and $r \in R$:

Initialization (Init): $\psi(I, I)$

Synchronized-step preservation (Sync):

$$\psi(a_1, a_2) \wedge Q(r) \wedge a_1' = f(a_1, r) \wedge a_2' = f(a_2, r) \;\Rightarrow\; \psi(a_1', a_2')$$

Stutter-step preservation (Stutter):

$$\psi(a_1, a_2) \wedge \neg Q(r) \wedge a_1' = f(a_1, r) \wedge a_2' = a_2 \;\Rightarrow\; \psi(a_1', a_2')$$

Final-state agreement (Final):

$$\psi(a_1, a_2) \;\Rightarrow\; \big(P(a_1) \wedge P'(a_2) \wedge a_1 = a_2\big) \vee \big(\neg P(a_1) \wedge \neg P'(a_2)\big)$$

Fig. 5. VCs for establishing that (Q, P') is a sound pushdown of P through UDF $F = \mathsf{fold}(x, I, f)$.

accumulators advance via the same accumulator f, and we must show that their updated internal states remain related by ψ. We refer to this verification condition as "synchronized-step preservation" (or Sync for short) because the two executions move in lockstep on this row.

Stutter-step preservation (Stutter). Our third condition addresses the case where $Q(r)$ is false: the optimized execution skips row r, while the original processes it. Here, the state in the original execution is updated via f, while the state in the optimized version remains unchanged. We must show that ψ continues to hold despite this asymmetry. The name "stutter step" (Stutter for short) reflects the fact that the optimized execution effectively "stutters" while the original moves forward.

Final-state agreement (Final). The final condition ensures that the bisimulation invariant ψ is strong enough to imply agreement between the two executions at the end. Specifically, *agreement* here means either (1) both executions satisfy their respective predicates, and their internal states are identical (i.e., $P(a_1) \wedge P'(a_2) \wedge a_1 = a_2$), or (2) both executions fail their respective predicates.

Definition 4.1. **(Certification Witness)** A predicate ψ is a *certification witness* for a candidate pushdown pair (Q, P') if ψ satisfies the verification conditions in Figure 5.

Example 4.2. We now illustrate our verification methodology in the context of the aggregation example from Section 2, where the `top2` UDF computes the two highest scores for each team. The original filter P requires that both the top and second-highest scores exceed 90.0:

$$P(a) = (a[0] > 90.0 \wedge a[1] > 90.0)$$

As discussed in Section 3, the optimal pushdown prunes the input and preserves behavior using

$$Q(r) = (r[\text{`score'}] > 90.0) \quad \text{and} \quad P'(a) = (a[1] \neq -\infty).$$

To verify the correctness of this pushdown optimization, we need the following bisimulation invariant ψ, where we use color to distinguish internal states from the original and optimized executions: blue refers to values computed over the full input (no filtering), and red refers to values computed after filtering by Q:

$$\underbrace{(a_1[0] > 90.0 \Rightarrow a_1[0] = a_2[0]) \land (a_2[0] > 90.0 \Rightarrow a_1[0] = a_2[0])}_{\text{(1) top-1 in sync if } > 90.0 \text{ in either}}$$
$$\land\ \underbrace{(a_1[1] > 90.0 \Rightarrow a_1[1] = a_2[1]) \land (a_2[1] > 90.0 \Rightarrow a_1[1] = a_2[1])}_{\text{(1) top-2 in sync if } > 90.0 \text{ in either}}$$
$$\land\ \underbrace{(a_2[0] \neq -\infty \Rightarrow a_2[0] > 90.0)}_{\text{(2) optimized top-1 undefined or } > 90} \land \underbrace{(a_2[1] \neq -\infty \Rightarrow a_2[1] > 90.0)}_{\text{(2) optimized top-2 undefined or } > 90.0} \land \underbrace{(a_2[1] > 90.0 \Rightarrow a_2[0] > 90.0)}_{\text{(3) top-2 } > 90.0 \text{ implies top-1 } > 90.0}$$

Intuitively, this invariant states that (1) whenever one of the two highest scores exceeds 90.0, it stays in sync between both executions (first four conjuncts); (2) throughout the optimized execution, each score is either undefined or exceeds 90.0 (next two conjuncts), and (3) if the second-highest score exceeds 90.0, the top score must too (last conjunct). Crucially, all of these conjuncts are necessary in order to prove the correctness of the pushdown optimization. Hence, as this example illustrates, bisimulation invariants in this setting can be quite complex, allowing for partial disagreement in internal states while still guaranteeing equivalence of the final outcomes.

We conclude this section by noting that our verification methodology is sound and relatively complete in the standard sense of Hoare-style logics [12]: whenever a candidate pushdown pair (Q, P') is correct and the underlying logic is sufficiently expressive, there exists a bisimulation invariant capable of establishing its correctness.

Theorem 4.3 (Soundness and Relative Completeness). *Let (Q, P') be a candidate solution to the generalized pushdown problem for computation F and predicate P. Then:*

(1) ***Soundness:*** *If there exists a bisimulation invariant ψ that is a certification witness for (Q, P') by Definition 4.1, then (Q, P') is a sound optimization, i.e., $\forall x.\ \mathsf{Lift}(P, F(x)) = \mathsf{Lift}(P', F(\mathsf{filter}_Q(x)))$.*

(2) ***Relative completeness:*** *If the above equivalence holds and the underlying logic is sufficiently expressive, then there exists an invariant ψ that is a certification witness for (Q, P').*

5 Synthesizing Optimal Pushdown Transformations

Given a computation F and a post-computation predicate P, our goal is to synthesize a triple (Q, P', ψ) such that (1) Q, P', and ψ satisfy the verification conditions from Figure 5, and (2) the solution is optimal according to Definition 3.4, meaning Q is as strong as possible, and P' is as weak as possible. If we can find such a triple, then (Q, P') can be used to obtain the best pushdown optimization for the original pipeline, while ψ serves as a *certificate of soundness* of the optimization. Hence, our synthesis method aims to simultaneously identify the best pushdown optimization possible, while also constructing its proof of correctness.

To motivate our solution, we first discuss the three main challenges in computing Q, P', and ψ. First, even ignoring the optimality requirement, this problem cannot be solved using off-the-shelf CHC solvers (as commonly done in program verification and predicate synthesis), because the VCs in Figure 5 are *non-Horn* due to the *negated* occurrence of Q in Stutter. Second, even if we manually supply the ground-truth solution for Q and thereby make the resulting VCs Horn, we still find that existing solvers struggle to infer ψ and P'. Finally, the optimality requirement makes an already-difficult inference problem even harder. In this context, Q and P' need to be optimized in *different* directions, creating a dual-criteria search in which we aim to find the strongest admissible Q and the weakest admissible P'.

Our synthesis method directly addresses these challenges through three core innovations. First, we introduce a *structured decomposition* of the search space that breaks down the joint inference of

Q, P', and ψ into a layered pipeline. By enumerating candidate input filters Q in decreasing order of strength, computing the strongest admissible bisimulation ψ for each, and then deriving the weakest consistent residual P', our method transforms a tangled three-way dependency into a tractable sequence of subproblems while still guaranteeing optimality. Second, we develop *symbolic bounds on bisimulation invariants* that act as *unrealizability certificates*. These bounds allow the synthesizer to quickly rule out entire families of pushdown candidates. Finally, our algorithm incorporates *root-cause-guided repair*, which analyzes the precise reason a verification condition fails and selectively refines the responsible artifact (i.e., Q, ψ, or P').

5.1 Top-Level Algorithm

Algorithm 1 shows our top-level synthesis procedure, SynthesizeOptimalPushdown, which seeks optimal instantiations of Q and P' expressible as conjunctions of atoms drawn from their respective predicate universes U_Q and $U_{P'}$. We deliberately confine the search to a fixed predicate universe for several reasons. First, real-world UDFs often manipulate complex internal state whose SMT encodings rely on algebraic data types, making automated refinement techniques such as Craig interpolation impractical in our setting. Moreover, static analysis of the UDF and its surrounding pipeline allows us to *a priori* extract the building blocks from which Q, P', and ψ can be formed. This includes not only simple literals but also richer predicates such as implications derived from control-flow structure. Finally, bounding synthesis to a finite universe yields a well-structured search space that can be systematically explored in decreasing order of logical strength for Q and increasing order for P', ensuring tractable search and clear optimality guarantees.

Given these predicate universes, Algorithm 1 starts by initializing a worklist with the strongest possible input-side filter (the conjunction of all atoms in U_Q) and explores progressively weaker candidates through the main loop (lines 3–11). In each iteration, it dequeues a candidate input-side filter Q_0 from the worklist and subjects it to a *feasibility check* by invoking WeakenViaBounds on line 4. If Q_0 is deemed infeasible (meaning that no sound pushdown optimization can exist with Q_0 as the input-side filter), WeakenViaBounds returns a repaired (weaker) version of Q_0, along with symbolic bounds $(\psi_{\min}, \psi_{\max})$ that summarize the feasible region for the target bisimulation invariant. These bounds act as under- and over-approximations for the true invariant ψ: That is, $\psi_{\min}$ collects atoms that *must* appear in any valid invariant, while $\psi_{\max}$ gathers those that *may*.

Once a repaired filter Q and symbolic bounds are obtained, the algorithm attempts to construct the strongest invariant ψ that is consistent with the bounds $(\psi_{\min}, \psi_{\max})$ by calling FindStrongestBisimulation (line 6). However, since the unrealizability check performed in WeakenViaBounds is sound but incomplete, Q *could* still be infeasible, causing FindStrongestBisimulation to fail. In this case, FindStrongestBisimulation also returns a *diagnosis* that identifies the specific verification condition violated by Q. This diagnosis takes one of two forms:

- **Sync**(r)**:** indicates that the *Sync* VC failed on row r. Intuitively, this means that if Q classifies r as selected (i.e., $Q(r)$ = true), then applying the accumulator function f to both executions would produce internal states that no bisimulation within $(\psi_{\min}, \psi_{\max})$ can reconcile. The only sound repair is therefore to *exclude* r by enforcing $Q(r)$ = false, implemented by removing all atoms in Q that evaluate to true on r.
- **Stutter**(r)**:** indicates that the *Stutter* VC failed on row r. In this case, Q classifies r as unselected (i.e., $Q(r)$ = false), but skipping r breaks agreement between the two internal states. Because no invariant can restore equivalence after this omission, the repair is to *force inclusion* of r by enforcing $Q(r)$ = true, i.e., by removing all atoms in Q that evaluate to false on r.

The algorithm then backtracks by enqueuing the repaired filter Q' along with all one-step weakenings of Q, each obtained by dropping a single atom. Otherwise, a valid invariant ψ is

Algorithm 1 SynthesizeOptimalPushdown

```
    Input: UDF F = fold(x, I, f), post-UDF filter P, predicate universes U_Q, U_P', U_ψ
    Output: Optimal pushdown pair (Q, P')
 1: function SynthesizeOptimalPushdown(F, P, U_Q, U_P', U_ψ)
 2:     W ← {U_Q}                                              ▷ Start from strongest filter
 3:     while Dequeue(W) = Some(Q_0) do
 4:         (ok, Q, ψ_min, ψ_max) ← WeakenViaBounds(Q_0, P, U_ψ)
 5:         if ¬ok then continue
 6:         (ok, ψ, diagnosis) ← FindStrongestBisimulation(Q, ψ_min, ψ_max)
 7:         if ¬ok then Q' ← Repair(Q, diagnosis); Enqueue(W, Q')
 8:         else
 9:             (ok, P') ← FindResidual(ψ, U_P', P)
10:             if ok then return (Q, P')
11:         for all q ∈ Q do Enqueue(W, Q \ {q})
12:     return Fail
13: function Repair(Q, diagnosis)
14:     if diagnosis = Sync(r) then return { u ∈ Q | ¬u(r) }          ▷ Force Q(r) = false
15:     else if diagnosis = Stutter(r) then return { u ∈ Q | u(r) }    ▷ Force Q(r) = true
```

successfully synthesized, and the algorithm proceeds to compute the weakest residual predicate P' by invoking FindResidual on line 9. If FindResidual succeeds, the algorithm terminates and returns (Q, P'); if not, it enqueues all one-step weakenings of Q and backtracks.

Discussion. A key point about this algorithm is that it can often establish the *infeasibility* of a candidate Q without attempting to synthesize either ψ or P'. Because each candidate Q could require exploring a vast space of bisimulation invariants and corresponding residuals, ruling out even a *single* candidate Q can be very beneficial. Furthermore, this pruning effect is amplified by the repair mechanism (via WeakenViaBounds), which implicitly removes from the search space *many* weakenings of Q that are infeasible for the same reason as Q. In other words, the infeasibility of a single Q can propagate to entire families of related filters, significantly reducing the search space (as shown empirically in Section 6).

Theorem 5.1 (Guarantees). *Fix finite predicate universes U_Q, $U_{P'}$, and U_ψ. For any computation F and post-filter P, SynthesizeOptimalPushdown$(F, P, U_Q, U_{P'}, U_\psi)$ satisfies the following properties:*

*(1) **Termination.** The procedure terminates.*
*(2) **Soundness.** If it returns (Q^*, P^*) where $Q^* \subseteq U_Q$ and $P^* \subseteq U_{P'}$, then (Q^*, P^*) satisfies Equation (3).*
*(3) **Optimality within the search space.** Let (Q^*, P^*) be the pair returned by the procedure. Then:*
  (a) (Maximal Q) For any $Q \subseteq U_Q$, $P' \subseteq U_{P'}$, and $\psi \subseteq U_\psi$ such that ψ is a certification witness for (Q, P'), Q is not strictly stronger than Q^*.
  (b) (Minimal P' given Q^*) For any $P' \subseteq U_{P'}$ and $\psi \subseteq U_\psi$ such that ψ is a certification witness for (Q^*, P'), P' is not strictly weaker than P^*.

Please note that Theorem 5.1 provides optimality only relative to the fixed universes U_Q, $U_{P'}$, and U_ψ. Accordingly, the procedure may miss valid pushdowns in two ways: (i) there may exist a valid decomposition (Q_0, P_0) with $Q_0 \nsubseteq U_Q$ or $P_0 \nsubseteq U_{P'}$ that is strictly better (stronger Q_0 or weaker P_0); or (ii) a valid pair (Q, P') with $Q \subseteq U_Q$ and $P' \subseteq U_{P'}$ may require a bisimulation invariant outside U_ψ. However, fixing these finite predicate universes is necessary for termination: Without restricting U_Q and $U_{P'}$, an optimum need not be finitely representable within the predicate language, and without restricting U_ψ, searching for an invariant witness does not admit termination

Algorithm 2 WeakenViaBounds

Input: Candidate input filter Q, post-UDF filter P, invariant universe U_ψ
Output: $(\text{ok}, Q', \psi'_{\text{min}}, \psi'_{\text{max}})$ where ok $\in$ {true, false}, potentially weakened filter Q', and potentially tightened symbolic bounds $(\psi'_{\text{min}}, \psi'_{\text{max}})$

```
1: function WeakenViaBounds(Q, P, U_ψ)
2:     (ψ_min, ψ_max) ← (∅, U_ψ)
3:     while true do
4:         if Q = ∅ then return (false, −, −, −)
5:         (ψ_min, ψ_max) ← RefineBounds(Q, P, U_ψ, ψ_min, ψ_max)
6:         (res, diagnosis) ← CheckUnrealizable(Q, ψ_min, ψ_max)
7:         if ¬res then return (true, Q, ψ_min, ψ_max) else Q ← Repair(Q, diagnosis)
```

guarantees in general. Additionally, solver timeouts or unknown answers may prevent, in practice, the procedure from finding an optimal solution with the fixed predicate universes.

5.2 Unrealizability Proofs via Symbolic Bounds on Bisimulation Invariants

Given a candidate filter Q, Algorithm 2 (WeakenViaBounds) is the engine that decides whether our algorithm should (a) keep Q as is, (b) *repair* it to a weaker predicate, or (c) abandon it completely. To make this decision, Algorithm 2 maintains a pair of symbolic bounds $(\psi_{\text{min}}, \psi_{\text{max}})$ over the invariant universe U_ψ that serve as symbolic lower and upper bounds on the bisimulation invariant.

WeakenViaBounds initializes $(\psi_{\text{min}}, \psi_{\text{max}})$ to the trivial bounds $(\varnothing, U_\psi)$ on line 2 and then gradually weakens the candidate filter Q until a termination condition is reached. In each iteration, the procedure alternates between *tightening* the symbolic bounds via RefineBounds (line 5) and querying CheckUnrealizable for an *unrealizability certificate* given the current bounds (line 6). These two procedures play complementary roles. RefineBounds addresses the question: *"Given the current Q, what is the tightest bound we can infer on the bisimulation invariant?"* On the other hand, CheckUnrealizable addresses the question: *"Given the current bounds $(\psi_{\text{min}}, \psi_{\text{max}})$, can we prove that no invariant ψ in this range will suffice to prove the correctness of Q?"* Together with the repair mechanism on line 7 (false branch), these two procedures form a virtuous cycle in the following way: First, the refined bounds may allow us to prove the unrealizability of a filter we previously could not prove to be infeasible. Second, once we detect infeasibility of Q, we can weaken it through targeted repair (via the same Repair procedure from earlier), which in turn allows us to further refine the bounds on ψ. This alternation between bound tightening and unrealizability checking continues until either (a) Q becomes $\varnothing$ and is deemed hopeless (line 4) or (b) Q appears to be consistent with the bounds (true branch on line 7). In the remainder of this subsection, we discuss the sub-procedures RefineBounds and CheckUnrealizable.

Computing symbolic bounds on the bisimulation invariant. Algorithm 3 (RefineBounds) maintains the symbolic bounds $(\psi_{\text{min}}, \psi_{\text{max}})$ within which any valid bisimulation must lie. The lower bound ψ_{min} is derived from Final and is computed only *once*, as this VC is independent of Q. One subtlety here is that Final also depends on P', which is completely unknown at this stage. To sidestep this dependency, RefineBounds instantiates occurrences of the unknown residual P' in the VC with the known filter P, yielding the formula ϕ on line 3 of Algorithm 3. Intuitively, P represents the strongest possible residual; thus, if the input-side filter Q is indeed valid, then applying Q in conjunction with the original filter P should already yield a behaviorally equivalent result. Hence, substituting P' with P in Final can only *weaken* the requirement and therefore produces a conservative approximation for deriving a lower bound on ψ. In particular, any implicant of formula

Algorithm 3 REFINEBOUNDS

Input: Candidate filter Q, post-filter P, invariant universe U_ψ, symbolic bounds $(\psi_{\min}, \psi_{\max})$
Output: Refined bounds $(\psi_{\min}, \psi_{\max})$

```
 1: function REFINEBOUNDS(Q, P, U_ψ, ψ_min, ψ_max)
 2:     if ψ_min = ∅ then                                   ▷ Lower bound (computed once)
 3:         ϕ ← (P(a1) ∧ P(a2) ∧ a1 = a2) ∨ (¬P(a1) ∧ ¬P(a2))
 4:         ψ_min ← FINDWEAKESTIMPLICANT(U_ψ, ϕ)
 5:     for all p ∈ ψ_max \ ψ_min do                        ▷ Upper bound at base case (computed once)
 6:         if ¬p(I, I) then ψ_max ← ψ_max \ {p}
 7:     for all p ∈ ψ_max \ ψ_min do                        ▷ Upper bound based on current Q
 8:         φ ← ψ_max(a1, a2) ∧ Q(r) ∧ a1' = f(a1, r) ∧ a2' = f(a2, r)
 9:         if φ ⇏ p(a1', a2') then ψ_max ← ψ_max \ {p}
10:     return (ψ_min, ψ_max)
```

ϕ from line 3 serves as a valid lower bound, but we compute the *weakest prime implicant* [38] to ensure that the bound is as tight as possible.[2]

In contrast, the upper bound $\psi_{\max}$ is refined incrementally as Q evolves because its computation involves Sync, which is dependent on Q. At initialization, $\psi_{\max}$ contains all atoms in the universe U_ψ. Each call to REFINEBOUNDS prunes this set in two ways. First, the algorithm removes any atom that fails Init.[3] Then, it checks whether each remaining atom p is preserved by Sync under the current $\psi_{\max}$. Specifically, it instantiates Sync with the full upper bound in the antecedent and the candidate atom p alone in the consequent:

$$\psi_{\max}(a_1, a_2) \wedge Q(r) \wedge a_1' = f(a_1, r) \wedge a_2' = f(a_2, r) \implies p(a_1', a_2').$$

If this implication fails, the atom p is removed from $\psi_{\max}$.

It is worth noting that the REFINEBOUNDS procedure does *not* use Stutter in refining the upper bound in order to ensure soundness. Specifically, recall that Stutter contains the negation of Q in its antecedent. Since we start from the strongest possible Q and gradually weaken it until it is sound, the current candidate Q could be logically stronger than the sound solution Q^*. This, in turn, means that the antecedent of Stutter could be weaker than when it is instantiated with the ground truth solution Q^*. Hence, it would be unsound to drop predicates that are not implied by the current antecedent. Due to this asymmetry, REFINEBOUNDS only uses Init and Sync.

To illustrate, consider the `top2` running example from Section 2 with $I = (-\infty, -\infty)$. The Init pass evaluates each atom on (I, I): for instance, $a_1[0] = -\infty \Rightarrow \text{false}$ ("the top score is always defined") fails and is discarded, while $a_1[0] = -\infty \Rightarrow a_1[0] = a_2[0]$ ("if undefined in the original, then also in the optimized") evaluates to $-\infty = -\infty$ and survives. The Sync pass then removes atoms incompatible with $Q(r) = (r[\text{'score'}] > 90.0)$: the atom $(a_1[0] > 90.0 \Rightarrow \text{false})$ survives Init but fails Sync, since processing a qualifying row forces the top score above 90.0.

Checking unrealizability via symbolic bounds. Algorithm 4 (CHECKUNREALIZABLE) uses the bounds $(\psi_{\min}, \psi_{\max})$ to decide whether the current filter Q could admit a sufficiently strong bisimulation. Rather than synthesizing an explicit invariant, it asks a simpler question: is *any* invariant lying between these bounds compatible with the one-step proof obligations? To answer this question, the procedure instantiates the two key verification conditions (namely, Sync and Stutter) using the bounds in their most conservative form: $\psi_{\max}$ is placed in the antecedent (the strongest possible

[2] See Appendix A for the definition of FINDWEAKESTIMPLICANT and its correctness proofs.
[3] This initialization check $p(I, I)$ only needs to be performed on the first invocation of REFINEBOUNDS.

Algorithm 4 CheckUnrealizable

Input: Candidate input filter Q, symbolic bounds $(\psi_{\min}, \psi_{\max})$
Output: Unrealizability judgment and root cause diagnosis: $\mathsf{Sync}(r)$ or $\mathsf{Stutter}(r)$

1: **function** CheckUnrealizable($Q, \psi_{\min}, \psi_{\max}$)
2: $\chi_{\mathsf{sync}} \leftarrow \big((\psi_{\max}(a_1, a_2) \land Q(r) \land a_1' = f(a_1, r) \land a_2' = f(a_2, r)) \implies \psi_{\min}(a_1', a_2')\big)$
3: **if** $\mathsf{Invalid}(\chi_{\mathsf{sync}})$ **then**
4: $(a_1, a_2, r) \leftarrow \mathsf{Model}(\neg\chi_{\mathsf{sync}})$
5: **return** $(\mathsf{true}, \mathsf{Sync}(r))$
6: $\chi_{\mathsf{stutter}} \leftarrow \big((\psi_{\max}(a_1, a_2) \land \neg Q(r) \land a_1' = f(a_1, r) \land a_2' = a_2) \implies \psi_{\min}(a_1', a_2')\big)$
7: **if** $\mathsf{Invalid}(\chi_{\mathsf{stutter}})$ **then**
8: $(a_1, a_2, r) \leftarrow \mathsf{Model}(\neg\chi_{\mathsf{stutter}})$
9: **return** $(\mathsf{true}, \mathsf{Stutter}(r))$
10: **return** $(\mathsf{false}, -)$

assumption) and $\psi_{\min}$ in the consequent (the weakest admissible conclusion). If *even this* implication fails, then no invariant consistent with the current candidate Q could possibly satisfy the condition.

If Sync fails, the procedure returns a witness $\mathsf{Sync}(r)$, where r is a symbolic row that demonstrates the violation. Intuitively, this means that if Q were to classify r as selected, the two internal states would be forced into a disagreement that no invariant between $(\psi_{\min}, \psi_{\max})$ could reconcile. The only possible repair is therefore to exclude r, i.e., to enforce $Q(r) = \mathsf{false}$. Dually, if Stutter fails, the procedure returns $\mathsf{Stutter}(r)$ with a witness r. In this case, the violation arises because skipping r (i.e., $Q(r) = \mathsf{false}$) would break agreement, and no admissible invariant could compensate. In this case, the only repair is to retain r by enforcing $Q(r) = \mathsf{true}$. If both checks succeed, the procedure returns $(\mathsf{false}, -)$, meaning that there is at least one invariant consistent with the current bounds that *could* establish the correctness of Q. Importantly, this is not a guarantee that such an invariant can actually be synthesized but merely that the current filter Q has not yet been refuted by the bounds. Thus, CheckUnrealizable acts as a sound but incomplete filter.

Returning to the `top2` example, suppose U_Q additionally contains $r[\text{‘score’}] > 95.0$. The Stutter check finds a unrealizability witness with $a_1 = a_2 = (97.0, -\infty)$ and $r[\text{‘score’}] = 92.0$: the optimized execution skips r (since $92.0 < 95.0$), but the original processes it, producing $a_1' = (97.0, 92.0)$ with $P(a_1') = \mathsf{true}$ while $P(a_2') = P(a_2) = \mathsf{false}$, violating $\psi_{\min}$. The repair forces $Q(r) = \mathsf{true}$ by dropping $r[\text{‘score’}] > 95.0$ (false on the witness), weakening Q to $r[\text{‘score’}] > 90.0$.

5.3 Synthesizing Bisimulation Invariants and Residuals

Given a candidate Q, recall that the top-level synthesis algorithm must construct two remaining artifacts: first, the strongest bisimulation invariant ψ consistent with Q, and *then* the weakest residual predicate P'. This ordering preserves optimality: for any given choice of P', if we cannot prove Final using the strongest possible bisimulation invariant, we certainly also cannot prove it using a weaker one. Thus, our algorithm first infers the strongest possible bisimulation invariant and then uses it to search for the weakest possible residual.

Algorithm 5 (FindStrongestBisimulation) computes the strongest bisimulation invariant for a given candidate Q. This is the most straightforward part of the algorithm and follows the familiar Houdini-style weakening loop [17], with two modifications. First, instead of initializing from the full invariant universe U_ψ, it uses the upper bound $\psi_{\max}$, which already excludes atoms ruled out by earlier analysis. Second, it enforces the symbolic lower bound $\psi_{\min}$ throughout. Whenever $\psi_{\min}$ is violated, we perform root cause analysis as follows: any attempt to remove an atom in $\psi_{\min}$

Algorithm 5 FindStrongestBisimulation

Input: Candidate input filter Q, symbolic bounds $(\psi_{\text{min}}, \psi_{\text{max}})$
Output: Strongest bisimulation invariant ψ or root cause diagnosis upon failure

```
1: function FindStrongestBisimulation(Q, ψ_min, ψ_max)
2:     ψ_cand ← ψ_max
3:     repeat until ψ_cand converges
4:         (ok, ψ_cand, r) ← WeakenViaVC(Q(r), f(a_2, r), ψ_min, ψ_cand)        ▷ Sync
5:         if ¬ok return (false, −, Sync(r))
6:         (ok, ψ_cand, r) ← WeakenViaVC(¬Q(r), a_2, ψ_min, ψ_cand)             ▷ Stutter
7:         if ¬ok return (false, −, Stutter(r))
8:     return (true, ψ_cand, −)
9: function WeakenViaVC(G(r), t_2, ψ_min, ψ_cand)
10:    for all p ∈ ψ_cand do
11:        χ ← ((ψ_cand ∧ G(r) ∧ a'_1 = f(a_1, r) ∧ a'_2 = t_2) ⇒ p(a'_1, a'_2))
12:        if Invalid(χ) then
13:            if p ∈ ψ_min then (a_1, a_2, r) ← Model(¬χ); return (false, −, r)
14:            else ψ_cand ← ψ_cand \ {p}
15:    return (true, ψ_cand, −)
```

indicates that no admissible invariant exists for the current filter, and the procedure returns false. In addition, it returns a symbolic row r as a witness to guide subsequent repair (lines 5 and 7).

Remark. This Houdini-style inference of the bisimulation invariant in Algorithm 5 is only possible because of our careful stratification of the search: Without a fixed Q, we would not be able to apply such an algorithm because the constraints governing ψ change along with Q.

Next, Algorithm 6 (FindResidual) describes how to compute the weakest residual for a given Q and ψ. At a high level, this algorithm iteratively strengthens the candidate residual until the Final VC becomes valid. As an initial optimization, Algorithm 6 checks whether the original filter P satisfies Final (lines 3–4). If not, Q is too strong, and no residual can succeed, so the procedure returns false. Otherwise, the algorithm considers increasingly stronger candidates and returns the current candidate P' as soon as Final becomes valid. The key part of the procedure is how it handles failures by performing root cause analysis in HandleFailure. If χ is violated, line 13 of Algorithm 6 leverages a model (a_1, a_2) of $\neg\chi$ to identify a possible repair strategy. In particular, the algorithm differentiates between three failure modes:

- **Mismatch on acceptance:** If $P(a_1) = P'(a_2) = \text{true}$ and $a_1 \neq a_2$ (line 16 in HandleFailure), the original and optimized executions disagree on which outputs they accept. No strengthening of the residual predicate can resolve this case, so the algorithm discards the current candidate.
- **False rejection:** If $P(a_1) = \text{true}$ and $P'(a_2) = \text{false}$ (line 17), the optimized execution incorrectly rejects an output that the original accepts. As in the previous case, strengthening P' cannot reverse this outcome, so the candidate is again discarded.
- **Spurious acceptance:** If $P(a_1) = \text{false}$ and $P'(a_2) = \text{true}$ (lines 18–20), the optimized execution incorrectly accepts an output that should be rejected. In this case, the algorithm strengthens P' to exclude a_2 by conjoining each atom $p \in U_{P'} \setminus P'$ such that $p(a_2) = \text{false}$, and enqueues each strengthened candidate onto the worklist W.

Importantly, the root cause analysis performed by HandleFailure further reduces the search space of the synthesis procedure, as not all failed residual candidates require further exploration.

Algorithm 6 FindResidual

Require: Bisimulation invariant ψ, residual universe $U_{P'}$, post-UDF filter P
Ensure: Weakest residual P' such that Final holds, or failure

```
1:  function FindResidual(ψ, U_P', P)
2:      # Early check: does P' = P satisfy Final?
3:      χ ← (ψ(a1, a2) ⇒ [(P(a1) ∧ P(a2) ∧ a1 = a2) ∨ (¬P(a1) ∧ ¬P(a2))])
4:      if Invalid(χ) then return (false, −)
5:      W ← {⊤};   Visited ← ∅                      ▷ Worklist of residual candidates
6:      while W ≠ ∅ do
7:          P' ← Dequeue(W)
8:          if P' ∈ Visited then continue
9:          Visited ← Visited ∪ {P'}
10:         # Check Final under ψ for current P'
11:         χ ← (ψ(a1, a2) ⇒ [(P(a1) ∧ P'(a2) ∧ a1 = a2) ∨ (¬P(a1) ∧ ¬P'(a2))])
12:         if Valid(χ) then return (true, P')
13:         HandleFailure(P, P', Model(¬χ), U_P', W)   ▷ May enqueue strengthened candidates
14:     return (false, −)
15: function HandleFailure(P, P', (a1, a2), U_P', W)
16:     if P(a1) ∧ P'(a2) ∧ a1 ≠ a2 then return      ▷ Mismatch on acceptance
17:     else if P(a1) ∧ ¬P'(a2) then return          ▷ False rejection
18:     else if ¬P(a1) ∧ P'(a2) then                 ▷ Spurious acceptance
19:         for all p ∈ U_P' \ P' do
20:             if ¬p(a2) then P'' ← P' ∧ p; Enqueue(W, P'')
```

6 Evaluation

We have implemented our method in a new tool called Pusharoo, which is written in OCaml and uses the Z3 SMT solver [13]. Pusharoo leverages a custom, Python-like DSL to represent UDFs in a uniform, framework-agnostic way. This DSL serves as a realistic intermediate representation (IR) for data pipelines written in Python, Scala, or Java, enabling Pusharoo to support a broad range of data processing frameworks such as pandas and Spark. To construct suitable predicate universes for each task, Pusharoo performs lightweight data-flow analyses over the UDF accumulator function f, initializer I, and post-UDF filter P. Concretely, Pusharoo constructs the predicate universes for Q, P', and invariants using a largely two-step procedure: First, it extracts a finite set of base predicates from each benchmark by taking the post-filter predicate and the UDF's control-flow conditions and normalizing them into CNF clauses. Then, it augments this base set of CNF clauses with disjunctive combinations between related atoms, where the notion of "related" is determined based on data dependence. Further details about our implementation, including the DSL and the predicate universe construction procedure, can be found in Appendix C.

In the rest of this section, we describe an experimental evaluation of Pusharoo that aims to answer the following research questions:

RQ1. How does our method compare against prior work [48]?
RQ2. How efficiently can Pusharoo synthesize correct predicate pushdown decompositions, and what is the complexity of the synthesized predicates and invariants?
RQ3. What impact do pushdown optimizations have on the running time of real-world pipelines?
RQ4. What is the impact of key design decisions on Pusharoo's synthesis efficiency?
RQ5. How sensitive is Pusharoo to the size of the predicate universe in terms of solution quality and synthesis time?

Benchmarks. To answer these questions, we evaluated Pusharoo on 150 benchmarks, consisting of 26 unique UDFs and 3-9 predicates per UDF. The UDFs used in our evaluation are drawn from two sources: (i) 19 Spark UDFs taken in their entirety from prior work [47] and (ii) 7 pandas UDFs crawled from GitHub using the same inclusion criteria as [47], focusing on UDFs that operate over the entire dataframe rather than row-wise maps and that involve at least one of conditional logic, collection processing, or multiple sub-aggregations. These benchmarks cover diverse domains (e.g., finance, machine learning, healthcare), and all are stateful in the fold sense: That is, each UDF updates an accumulator state across the input sequence, so they are not row-wise stateless map functions. The pandas UDFs are written in Python, while the Spark UDFs use Scala or Java.

Pre-processing. Since these benchmarks span several languages, we manually translated them to Pusharoo's underlying DSL. This translation is largely syntax-directed and preserves the structure of the original computation. In a few cases, our translation required making some semantics-preserving adaptations, such as replacing timestamps with integers, to align with our DSL.

Table 1. UDF statistics.

Feature	**Value**
Total UDFs	26
pandas UDFs	7
Spark UDFs	19
# w/ conditionals	23
# w/ collections	8
# w/ tuple accumulator	18
# w/ cross-dependent aggregations	8
Avg AST size	77
Max AST size	168

UDF statistics. Table 1 summarizes key structural and syntactic features of the UDFs used in our evaluation. The vast majority (85%) include conditionals, 31% utilize collections, and 69% use tuple-valued accumulators to track multiple pieces of state. Furthermore, 8 of the 26 UDFs (31%) feature cross-dependent aggregations, where the computation of one aggregation depends on the intermediate results of one or more of others. These characteristics, along with nontrivial AST sizes (average 77 nodes), highlight the semantic complexity of the benchmarks.

Filters. To evaluate Pusharoo's pushdown capabilities, each benchmark must be paired with a post-UDF filtering predicate. For benchmarks that already include a post-UDF filter (as is the case for most `pandas` cases), we retained the original predicate. To broaden the evaluation space, we also generated additional predicates using a schema designed to reflect common filtering idioms:

- **Scalar-valued outputs:** For UDFs that return a single scalar value v, we generated predicates of the form $\{v \ \texttt{op} \ c \mid \texttt{op} \in O, c \in C\}$, where $O = \{=, \neq, >, \geq, <, \leq\}$ and C is a set of constants either drawn from the original pipeline or chosen to reflect typical threshold-style filters. For cases where v is an `Optional` value, we wrapped the predicate in a `match` expression, with the `Some` branch applying the comparison and the `None` branch returning a fixed Boolean.
- **Tuple-valued outputs:** For UDFs that return a tuple a, we constructed field-level predicates of the form $\{a[i] \ \texttt{op} \ c \mid 0 \leq i < \texttt{length}(a), \texttt{op} \in O, c \in C\}$, applying the same treatment as in the scalar case for any `Optional` fields. These serve as building blocks for more complex filters.
- **Compound predicates:** To capture more expressive queries, we composed field-level predicates into compound filters using conjunctions and disjunctions across multiple tuple fields. These reflect common patterns of multi-attribute filtering found in analytic workloads.

For each UDF, we generated 10–20 candidate predicates. Since our goal is to evaluate Pusharoo's optimization capability when pushdown is actually feasible, we retained only those predicates offering genuine optimization opportunities. We first checked pushdown feasibility on small, bounded symbolic dataframes (see Appendix B for details) and kept only benchmarks where pushdown was *potentially* feasible. We then manually filtered these cases to ensure that optimization was indeed possible. After this two-stage filtering, each UDF contributed between three and nine valid predicates, with an average of 5.8, yielding a total of 150 benchmarks for evaluation.

Table 2. Pushdown outcomes for Pusharoo vs. MagicPush's theoretical best-case results. The "Any" column reports the number of benchmarks for which each tool successfully applies *some* form of pushdown.

Type	# Benchmarks	Pusharoo				MagicPush (Upper Bound)		
		Any	Exact	Partial	Split	Any	Exact	Partial
pandas	27	27 (**100.0%**)	9 (**33.3%**)	12 (**44.4%**)	6 (22.2%)	12 (44.4%)	5 (18.5%)	7 (25.9%)
Spark	123	123 (**100.0%**)	2 (**1.6%**)	40 (**32.5%**)	81 (65.9%)	10 (8.1%)	0 (0.0%)	10 (8.1%)
Overall	150	150 (**100.0%**)	11 (**7.3%**)	52 (**34.7%**)	87 (58.0%)	22 (14.7%)	5 (3.3%)	17 (11.3%)

Experimental setup. All of the experiments are conducted on a machine with an Apple M3 Max chip (14-core CPU) and 36 GB of memory. We use a 10-minute timeout for each benchmark.

6.1 Comparison with Prior Work

In this section, we compare Pusharoo to MagicPush [48], the only existing approach that performs predicate pushdown for UDFs. We first provide some necessary background on MagicPush and then present our empirical findings.

Background. MagicPush supports only exact and partial pushdown and verifies each candidate using a small-model theorem. It performs bounded symbolic execution on a fixed "mini-table," then relies on structural preconditions to lift the proof to unbounded inputs. These preconditions require that the UDF: (1) be associative and commutative; (2) admit a "small-model" reduction, i.e., a smaller input yielding the same output; (3) be monotonic with respect to the post-UDF predicate; and (4) be insensitive to rows removed by the pre-filter. To synthesize pre-UDF filters, MagicPush enumerates disjunctions of predicate conjunctions drawn from the pipeline. Each candidate is checked for *exact* pushdown; if this fails, it is re-evaluated for *partial* pushdown.

Experimental setup. Since MagicPush's complete implementation is not publicly available, we cannot evaluate it directly.[4] Instead, we conservatively report an *upper bound* on MagicPush's performance: for any benchmark that meets its structural preconditions and passes the bounded check, we assume it could perform exact or partial pushdown. However, this estimate is highly optimistic: since MagicPush's search space grows combinatorially, it may still fail to find a valid pushdown within practical time limits even when these conditions hold.

Main results. Table 2 summarizes the evaluation results. For Pusharoo, we distinguish exact pushdowns (no residual check), partial pushdowns (residual identical to the original post-UDF filter), and split pushdowns (residual strictly weaker). For MagicPush, "Exact" and "Partial" count benchmarks whose structural preconditions and bounded check permit those modes; MagicPush does not support split pushdown.

Two main trends stand out in Table 2. First, Pusharoo successfully applies some form of pushdown to *every* benchmark in our suite, while MagicPush applies to only 22 out of 150 cases (14.7%), underscoring Pusharoo's broader applicability. Second, Pusharoo's generalized split pushdown dominates both exact and partial modes across the board, demonstrating that many UDFs admit nontrivial residuals that go beyond exact and partial pushdown.

Optimality. By design, Pusharoo always returns the strongest pre-filter Q^* and weakest residual P^* within our chosen predicate universe, so every solution is optimal in that sense. Since proving optimality over all conceivable predicates lies beyond our finite search space, Pusharoo does not provide a global optimality guarantee. Hence, to evaluate optimality in practice, we manually

[4]The implementation of MagicPush was not publicly released alongside the original paper [48]. We contacted the authors at the start of this work and received partial source code. However, essential components of their tool and benchmark data were unavailable and could not be recovered, making a direct empirical comparison infeasible.

Table 3. Breakdown of the types of pushdowns synthesized by Pusharoo.

Type	# Benchmarks	Avg Runtime (s)	Avg $\|Q^*\|$	Avg $\|P^*\|$	Avg $\|\psi\|$	Avg $\|P\|$
Exact	11	0.26	1.18	N/A	11.27	1.18
Partial	52	1.98	2.52	4.60	19.69	5.48
Split	87	5.30	2.31	3.41	36.91	5.09
Overall	150	3.78	2.30	3.57	29.06	4.94

examined 10 benchmarks selected by taking two or three examples from each of the four UDF categories from Table 1, spanning a range of UDF and post-filter complexities. For each selected benchmark, we manually constructed what we believed to be the optimal pushdown and compared it against the decomposition synthesized by Pusharoo. In all 10 cases, the synthesized result matched the manually identified optimum. Furthermore, among the 17 cases where MagicPush can theoretically apply a partial pushdown optimization, Pusharoo finds a weaker (and simpler) residual than the original post-UDF predicate for 8 of these (47.1%). These findings indicate that, even in cases where MagicPush is applicable, Pusharoo produces strictly better decompositions.

RQ1 Summary: Pusharoo is more expressive than MagicPush, achieving valid pushdown transformations on **100%** of our 150 benchmarks. In contrast, MagicPush can feasibly be applied to only **22/150 (14.7%)** of these cases, leaving **85.3%** of real-world UDFs beyond its theoretical capabilities.

6.2 Running Time and Predicate Complexity

In this section, we evaluate Pusharoo's performance in terms of synthesis time and report on the complexity of the synthesized artifacts. The results of this evaluation are presented in Table 3, which groups benchmarks by solution type: exact, partial, or split. For each category, we report the number of benchmarks, average synthesis time, and average sizes of the strongest pre-filter Q^*, the weakest residual P^*, the original predicate P, and the bisimulation invariant ψ.

Overall, Pusharoo synthesizes pushdowns in less than 4 seconds on average. The synthesized pre-filters have an average size of 2.3 clauses, and residual checks average 3.6 clauses compared to 4.9 for the original predicates. Bisimulation invariants average 29.1 in size, indicating that proving correctness of pushdown optimizations over UDFs is often nontrivial. As expected, split decompositions incur the highest synthesis cost (5.30s on average) as they require finding a more complex invariant with average size 36.9. Exact cases, which eliminate the residual entirely, complete fastest (0.26s on average) and have the simplest invariants (11.3 in size). Partial cases fall in between, taking less than 2 seconds and requiring invariants of average size 19.7. This progression reflects the deeper semantic reasoning required as solutions move from exact to partial to split. Appendix D presents two case studies to illustrate the kind of pushdowns enabled by Pusharoo.

RQ2 Summary: Pusharoo takes a median of 1.60s (average 3.78s) to synthesize optimal pushdown optimizations. The generated bisimulation invariants have an average size of 29.1, reflecting the complexity of the relational reasoning required. Finally, on 87 of 150 benchmarks, Pusharoo produces split solutions that go beyond exact and partial pushdown.

6.3 End-To-End Performance

Beyond synthesis coverage and efficiency, we also measured the run-time impact of our pushdown optimizations on the 150 pandas and Spark pipelines by comparing end-to-end runtimes before and after optimization. Following empirical benchmarking practices common in database research [14,

26, 36], we generated 100 million input rows for each pipeline using mixed Zipfian and uniform distributions chosen to mirror realistic domain semantics. As shown in Table 4, Pusharoo reduced execution time by 58.8% on average (median 61.4%), with some pipelines achieving improvements of up to 99.2%. These savings correspond to an average speedup of 2.4×, with extreme cases reaching two orders of magnitude. As expected, exact pushdown yields the largest speedup on average by completely eliminating the post-UDF filter P, followed by split pushdown, which weakens P, and finally partial pushdown, which preserves it.

Table 4. Pipeline runtime reduction.

Type	Min	Avg	Median	Max
Exact	36.9%	70.1%	80.8%	91.4%
Partial	24.2%	56.1%	54.7%	84.2%
Split	25.1%	59.1%	61.8%	99.2%
Overall	24.2%	58.8%	61.4%	99.2%

Split vs. partial pushdown. To quantify the benefit of synthesizing a weakened residual over retaining the original post-UDF filter, we compared split pushdown against partial pushdown on the 87 benchmarks that admit split pushdown. Across these, split pushdown is on average 5.1% faster, up to 14%. The gains are most pronounced for UDFs returning complex types (e.g., large tuples or collections), where redundant predicate evaluation is more expensive.

RQ3 Summary: Pusharoo optimizations yield major end-to-end runtime savings, speeding up execution by 2.4× on average and up to two orders of magnitude in some cases.

6.4 Ablation Studies

To evaluate the significance of key algorithmic choices in Pusharoo, we conducted ablation studies by disabling or replacing essential components individually. Specifically, we consider:

- Pusharoo-NoBounds, which does not infer bounds on the bisimulation invariant.
- Pusharoo-NoRepair, which disables all root cause analyses and targeted repairs.
- Pusharoo-TwoPhase, which first generates candidate filters Q, P' and *then* verifies their correctness by attempting to find a bisimulation invariant.
- Pusharoo-NoPrune, which enumerates Q candidates from strongest to weakest, performing an independent full solve for each without sharing invariant reasoning across candidates.
- Pusharoo-Spacer and Pusharoo-Eldarica, which replace lines 4–10 of SynthesizeOptimal-Pushdown (Algorithm 1) with a call to Spacer [24] and Eldarica [22] (two of the best-performing CHC solvers), respectively, to infer both ψ and P'.

We note that the Pusharoo-Spacer and Pusharoo-Eldarica ablations use an off-the-shelf CHC solver to infer *only* the bisimulation invariant ψ and residual P', but not the pre-UDF filter Q. By instantiating Q before invoking the solver, all VCs remain in strict Horn form (i.e., each clause has at most one positive head literal) so Spacer and Eldarica can, in principle, solve the resulting Horn clauses. If Q is instead treated as an unknown, the Horn-clause restriction is violated, causing CHC solvers to fail outright. Additionally, we note that Pusharoo-Spacer and Pusharoo-Eldarica do not have optimality guarantees and may not find the weakest residual.

Solve rate and running time comparison. To quantify the impact of these ablations on solve rate and running time, we present a *cumulative distribution function (CDF) plot* in Figure 6 that shows the percentage of benchmarks solved (y-axis) against per-benchmark solving time (x-axis).

Figure 6 shows that Pusharoo significantly outperforms all ablations. Among all variants, Pusharoo-TwoPhase performs the worst, highlighting the importance of the tight coupling between synthesis (i.e., finding pushdown predicates) and verification (i.e., finding a bisimulation to prove correctness of optimization). Even the best performing ablation, namely Pusharoo-NoBounds, solves fewer benchmarks and takes 9.6× as long to solve the benchmarks that can be solved by both Pusharoo and Pusharoo-NoBounds.

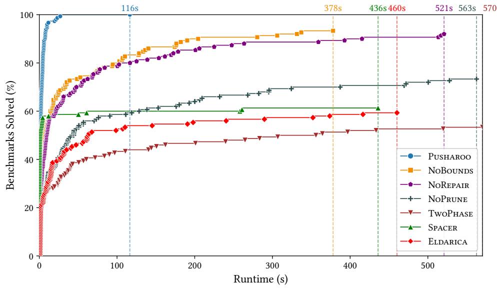

Fig. 6. CDF plot of % of benchmarks solved over solving time for Pusharoo and ablations.

Table 5. Search space % reduction (median / average / max) enabled by Pusharoo.

Type	NoBounds	NoRepair	TwoPhase
Exact	33.3 / 36.4 / 50.0	33.3 / 37.9 / 50.0	0.0 / 6.1 / 66.7
Partial	52.6 / 51.6 / 94.4	50.0 / 57.3 / 99.2	85.3 / 59.1 / 99.4
Split	40.3 / 38.3 / 92.4	70.0 / 64.7 / 95.1	80.9 / 49.2 / 99.5
Overall	41.1 / 42.8 / 94.4	61.7 / 60.1 / 99.2	64.8 / 46.8 / 99.5

Optimality comparison. In addition to performing significantly worse than Pusharoo, the CHC ablations also do not come with optimality guarantees. To quantify this aspect, Table 6 additionally explores how often the CHC ablations yield a suboptimal decomposition compared to Pusharoo. Overall, Pusharoo-Spacer produces suboptimal residuals on 61 benchmarks (66% of what it solves), and Pusharoo-Eldarica on 51 benchmarks (57% of what it solves). These results demonstrate that replacing our core synthesis procedure with off-the-shelf CHC solvers not only causes many failures and timeouts, but also leads to suboptimal solutions for the majority of the solved benchmarks.

Table 6. Solution optimality.

Variant	Solved	Subopt
Pusharoo	150	0
Spacer	92	61
Eldarica	89	51

Search space reduction. While Figure 6 shows that our three algorithmic innovations (namely, structured search, unrealizability certificates, and targeted repair) have a significant impact on running time, we further directly measure the search space reduction that these ideas enable. As shown in Table 5, all of our algorithmic contributions result in a significant reduction in search space, with averages ranging from 43 to 60% across the three ablations.

RQ4 Summary: All of our core algorithmic ideas have a significant impact on synthesis time and success rate. Furthermore, the ablations that replace parts of the algorithm with off-the-shelf CHC solvers are also significantly worse and yield suboptimal solutions.

6.5 Sensitivity to Predicate Universe Size

To evaluate universe-size sensitivity, we systematically enlarge U_Q and U_ψ with new disjunctions of atoms while leaving $U_{P'}$ untouched: since the goal of weakening P is to simplify the post-UDF check by *dropping* conjuncts, adding disjunctions there does not make sense from the perspective of reducing post-UDF work. Table 7 shows the results. Adding predicates does not improve solution quality on any benchmark, suggesting our universe-construction heuristics already effectively capture relevant predicates. Furthermore, although worst-case complexity is exponential in $|U_Q|$, synthesis time grows modestly, indicating that our algorithmic optimizations are effective in practice.

Table 7. Sensitivity analysis.

Enlarged	By	% Better Solns	Avg Δ Time
U_Q	10%	0%	+2.0%
	20%	0%	+1.6%
	30%	0%	+3.5%
U_ψ	10%	0%	+9.7%
	20%	0%	+21.4%
	30%	0%	+29.2%
Both	10%	0%	+10.6%
	20%	0%	+21.2%
	30%	0%	+30.8%

RQ5 Summary: Enlarging the predicate universes does not improve solution quality, and synthesis time grows modestly rather than exhibiting the exponential blow-up suggested by worst-case complexity.

7 Related Work

Predicate pushdown. Predicate pushdown is a widely used optimization for improving query performance, supported by nearly all modern data systems including PostgreSQL, MySQL, SQL Server, Oracle, Spark, and Flink [8, 29–31, 37, 40]. Prior research primarily targets relational operators, built-in aggregations, or semi-structured data formats [6, 23, 27, 32, 33, 50, 56], but remains ineffective for most UDFs. To our knowledge, MagicPush [48] is the only prior system that performs pushdown through UDFs; however, it relies on restrictive assumptions that fail for the vast majority of the real-world benchmarks from Section 6. Additionally, MagicPush can only perform exact and partial pushdown and cannot infer weaker nontrivial residuals.

Optimizing UDFs. Many systems attempt to expose UDF semantics to query optimizers by translating imperative code into relational form. Representative approaches use program synthesis (Qbs [9]), static analysis (EqSQL [15], Aggify [21]), quey rewrites [41], and compiler-level inlining (Froid [39]). Recent work extends these ideas to Spark SQL and RDDs [51, 52], while Dias [4] applies lightweight rewrites to pandas workloads involving row-wise UDFs. Other directions include fusing multiple UDFs to avoid redundant computation [43] and verifying homomorphism properties of aggregations to synthesize merge operators for incremental and parallel execution [47].

Optimal synthesis. Our method relates to optimal synthesis techniques that produce correct solutions under logical or quantitative criteria, including OptPCSat [20] for CHC solving, multi-abduction [2] for weakest specification inference, loud [35] for extremal over and under approximations, and systems such as Synapse [7] and abstract-interpretation-guided approaches [28] that optimize cost metrics. However, these approaches do not address our setting, which requires synthesizing a pair of predicates with opposing optimality objectives (strongest Q and weakest P').

Relational verification. This paper is related to a long line of work on *relational verification* which aims to reason about the relationship between multiple programs or different executions of the same program. Existing relational verification techniques include relational program logics [1, 5, 42, 49], product programs [3], and methods for regression verification [16, 19, 25] In the database domain, several systems target SQL query equivalence [10, 11, 45, 57], and Mediator [46] addresses equivalence of database-driven applications under schema changes. In contrast to prior work, our goal is to simultaneously synthesize and verify the correctness of pushdown optimizations.

8 Conclusion

This paper develops a unified semantic framework for pushdown optimizations, which aim to reduce the amount of data a computation must process. Our framework generalizes and extends prior exact and partial pushdown transformations, and enables them on a broader class of computations, including those with complex internal state. Building on this foundation, we introduce a bisimulation-based verification method and a synthesis algorithm that constructs optimal pushdown transformations together with correctness proofs. The synthesis procedure solves a challenging second-order constraint solving problem via structured decomposition (to guarantee optimality), symbolic bounds on the bisimulation invariant (to prove unrealizability), and root-cause-guided predicate repair (for goal-directed reasoning). Our implementation, Pusharoo, applies these techniques to real-world pandas and Spark pipelines, synthesizing provably correct pushdown transformations in seconds and significantly outperforming the prior state-of-the-art, leading to substantial end-to-end improvements of up to two orders of magnitude.

Data-Availability Statement

The appendices can be found in the extended version of this paper [53] and Supplementary Material. The software artifact accompanying this paper, comprising the implementation of Pusharoo, the benchmark suite of 150 pandas and Spark pipelines, and scripts to reproduce the evaluation results in Section 6, can be found on GitHub [55] and Zenodo [54].

Acknowledgments

We thank the anonymous reviewers for their thoughtful feedback and suggestions. This work was conducted in a research group supported by NSF awards CCF-1918889, CNS-2120696, CCF-2210831, and CCF-2319471, CCF-2422130, CCF-2403211, CCF-2505865, CCF-2326576, and CCF-2403211, as well as a DARPA award under agreement HR00112590133 and a gift from Amazon.

References

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proc. ACM Program. Lang.* 1, ICFP, Article 21 (Aug. 2017), 29 pages. doi:10.1145/3110265

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. *SIGPLAN Not.* 51, 1 (Jan. 2016), 789–801. doi:10.1145/2914770.2837628

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *Proceedings of the 17th International Conference on Formal Methods* (Limerick, Ireland) *(FM'11)*. Springer-Verlag, Berlin, Heidelberg, 200–214. doi:10.1007/978-3-642-21437-0_17

[4] Stefanos Baziotis, Daniel D. Kang, and Charith Mendis. 2024. Dias: Dynamic Rewriting of Pandas Code. *Proc. ACM Manag. Data* 2, 1 (2024), 58:1–58:27. doi:10.1145/3639313

[5] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not.* 39, 1 (Jan. 2004), 14–25. doi:10.1145/982962.964003

[6] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986.* 61–71. doi:10.1145/16894.16861

[7] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 775–788. doi:10.1145/2837614.2837666

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[9] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 3–14. doi:10.1145/2491956.2462180

[10] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1482–1495. doi:10.14778/3236187.3236200

[11] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 510–524. doi:10.1145/3062341.3062348

[12] Stephen A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (Feb. 1978), 70–90. doi:10.1137/0207005

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3376–3388. doi:10.14778/3484224.3484234

[15] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of*

Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1781–1796. doi:10.1145/2882903.2882926

[16] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 349–360. doi:10.1145/2642937.2642987

[17] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, Berlin, Heidelberg, 500–517. doi:10.1007/3-540-45251-6_29

[18] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

[19] Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California) *(DAC '09)*. Association for Computing Machinery, New York, NY, USA, 466–471. doi:10.1145/1629911.1630034

[20] Yu Gu, Takeshi Tsukada, and Hiroshi Unno. 2023. Optimal CHC Solving via Termination Proofs. *Proc. ACM Program. Lang.* 7, POPL, Article 21 (Jan. 2023), 28 pages. doi:10.1145/3571214

[21] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 559–573. doi:10.1145/3318464.3389736

[22] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–7. doi:10.23919/FMCAD.2018.8603013

[23] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. 2011. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.* 4, 6 (2011), 385–396. doi:10.14778/1978665.1978670

[24] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 17–34. doi:10.1007/978-3-319-08867-9_2

[25] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 712–717. doi:10.1007/978-3-642-31424-7_54

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[27] Yiming Lin and Sharad Mehrotra. 2024. PLAQUE: Automated Predicate Learning at Query Time. *Proc. ACM Manag. Data* 2, 1 (2024), 46:1–46:25. doi:10.1145/3639301

[28] Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2024. Optimal Program Synthesis via Abstract Interpretation. *Proc. ACM Program. Lang.* 8, POPL, Article 16 (Jan. 2024), 25 pages. doi:10.1145/3632858

[29] Microsoft. 2025. Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server. Accessed June 2025.

[30] MySQL. 2025. MySQL. https://www.mysql.com/. Accessed June 2025.

[31] Oracle. 2025. Oracle Database. https://www.oracle.com/database/. Accessed June 2025.

[32] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265. doi:10.14778/3368289.3368292

[33] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before you parse: faster analytics on raw data with sparser. *Proc. VLDB Endow.* 11, 11 (July 2018), 1576–1589. doi:10.14778/3236187.3236207

[34] David Park. 1981. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, Peter Deussen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–183. doi:10.1007/BFb0017309

[35] Kanghee Park, Xuanyu Peng, and Loris D'Antoni. 2025. LOUD: Synthesizing Strongest and Weakest Specifications. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 114 (April 2025), 28 pages. doi:10.1145/3720470

[36] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 582–587. doi:10.1145/564691.564759

[37] PostgreSQL. 2025. PostgreSQL. https://www.postgresql.org/. Accessed June 2025.

[38] Willard V Quine. 1959. On cores and prime implicants of truth functions. *The American Mathematical Monthly* 66, 9 (1959), 755–760. doi:10.1080/00029890.1959.11989404

[39] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (2017), 432–444. doi:10.1145/3186728.3164140

[40] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164. doi:10.1007/s41060-016-0027-9

[41] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 532–543. doi:10.1109/ICDE.2014.6816679

[42] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 57–69. doi:10.1145/2908080.2908092

[43] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of queries with user-defined functions. *ACM SIGPLAN Notices* 49, 6 (2014), 554–564. doi:10.1145/2594291.2594305

[44] Jeffrey D. Ullman. 1990. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies.* W. H. Freeman & Co., USA.

[45] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (July 2024), 3602–3614. doi:10.14778/3681954.3682024

[46] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (Dec. 2017), 29 pages. doi:10.1145/3158144

[47] Ziteng Wang, Ruijie Fang, Linus Zheng, Dixin Tang, and Işı l Dillig. 2025. Homomorphism Calculus for User-Defined Aggregations. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 294 (oct 2025), 27 pages. doi:10.1145/3763072

[48] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2, Article 136 (June 2023), 28 pages. doi:10.1145/3589281

[49] Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1–3 (April 2007), 308–334. doi:10.1016/j.tcs.2006.12.036

[50] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113. doi:10.14778/3476249.3476265

[51] Guoqiang Zhang, Benjamin Mariano, Xipeng Shen, and Işıl Dillig. 2023. Automated Translation of Functional Big Data Queries to SQL. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 95 (April 2023), 29 pages. doi:10.1145/3586047

[52] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 112 (Oct. 2021), 26 pages. doi:10.1145/3485489

[53] Robert Zhang, Eric Hayden Campbell, Dixin Tang, and Isil Dillig. 2026. Optimal Predicate Pushdown Synthesis. arXiv:2604.13351 [cs.PL] https://arxiv.org/abs/2604.13351

[54] Robert Zhang, Eric Hayden Campbell, Dixin Tang, and Isil Dillig. 2026. *Optimal Predicate Pushdown Synthesis (Software Artifact).* doi:10.5281/zenodo.19688701

[55] Robert Zhang, Eric Hayden Campbell, Dixin Tang, and Isil Dillig. 2026. Pusharoo. https://github.com/utopia-group/pushdown

[56] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2021. SIA: Optimizing Queries using Learned Predicates. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2169–2181. doi:10.1145/3448016.3457262

[57] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2735–2748. doi:10.1109/ICDE53745.2022.00250

Received 2025-11-13; accepted 2026-04-03