Predicate Pushdown for Data Science Pipelines

CONG YAN, Microsoft Research, USA
YIN LIN[*], University of Michigan, USA
YEYE HE, Microsoft Research, USA

Predicate pushdown is a widely adopted query optimization. Existing systems and prior work mostly use pattern-matching rules to decide when a predicate can be pushed through certain operators like join or groupby. However, challenges arise in optimizing for data science pipelines due to the widely used non-relational operators and user-defined functions (UDF) that existing rules would fail to cover. In this paper, we present MagicPush, which decides predicate pushdown using a search-verification approach. MagicPush searches for candidate predicates on pipeline input, which is often not the same as the predicate to be pushed down, and verifies that the pushdown does not change pipeline output with full correctness guarantees. Our evaluation on TPC-H queries and 200 real-world pipelines sampled from GitHub Notebooks shows that MagicPush substantially outperforms a strong baseline that uses a union of rules from prior work – it is able to discover new pushdown opportunities and better optimize 42 real-world pipelines with up to 99% reduction in running time, while discovering all pushdown opportunities found by the existing baseline on remaining cases.

CCS Concepts: • **Theory of computation → Program verification**; • **Information systems → Query languages for non-relational engines**; **Query optimization**.

Additional Key Words and Phrases: query optimization, verification, user-defined functions

ACM Reference Format:
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. https://doi.org/10.1145/3589281

1 INTRODUCTION

Pipelines such as data science pipelines and ETL pipelines are more and more prevalent nowadays due to emerging applications of machine learning and data-driven business intelligence. These pipelines are processing a growing amount of data, and efficiency becomes crucial while challenging. It is tempting to apply well-studied optimizations in relational data processing to data science pipelines, yet challenges arise due to two major differences. First, data science pipelines often contain non-relational operators, including spread-sheet operators like pivot and unpivot, and schema-changing operators like drop-column and change-column-type. Second, user-defined functions (UDF) are widely used, particularly in systems where pipelines are developed in languages like Python or R with rich syntax and abundant libraries. Simply applying SQL query optimizations to data science pipelines would not be optimal as these non-relational operators and UDFs will be skipped while very often they can be optimized together with relational operators.

[*]Work done at Microsoft Research.

Authors' addresses: Cong Yan, cong.yan@microsoft.com, Microsoft Research, Redmond, USA; Yin Lin, irenelin@umich.edu, University of Michigan, Ann Arbor, USA; Yeye He, yeyehe@microsoft.com, Microsoft Research, Redmond, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/6-ART136 $15.00
https://doi.org/10.1145/3589281

In this paper, we focus on one well-known optimization, predicate pushdown, and explore how to achieve it in data science pipelines with non-relational operators and UDFs. Predicate pushdown moves predicates to where the data lives in order to filter out data earlier rather than later. It can optimize a query by filtering data before it is transferred over the network, skipping reading entire files or chunks of files, or filtering data before loading into memory. Furthermore, query processing can also be potentially accelerated with less data to be processed. Prior research and existing databases widely apply this optimization, achieving query speedup as much as 65× [32].

When a predicate **can be pushed down** is often decided by pattern-matching rules in relational databases. For instance, predicates involving columns from one table can be pushed through inner join, and predicates on grouping columns can be pushed down through group-by, etc. However, these rules are pattern-matching operators with predicates, far from sufficient for data science pipelines due to non-relational operators and UDFs: very often a predicate can be pushed through an operator/UDF, but such optimization opportunity is missed as existing rules do not cover this particular pattern. Furthermore, an explosion of new rules would be required to handle the variety of non-relational operators and the rich syntax of predicates/UDFs in data science pipelines, making the rule-based approach hard to scale and manage.

In this work, we propose using a search-and-verification method instead of designing more pattern-matching rules. The main idea is to enumerate predicates on the input table and then use symbolic execution [60] to verify whether a pushdown is correct. Symbolic execution runs queries on tables whose cell values are symbols instead of concrete values, then verifies the equivalence of query results before and after predicate pushdown. It can handle arbitrary predicates, operators, and UDFs as it essentially runs two programs and compares their output. This technique has been widely used in prior work, like showing the inequivalence of SQL queries with counter examples [27], checking the correctness of new non-relational query plans [66] or optimized Spark queries [56], etc.

However, the major limitation of this approach is that the verification is bounded: symbolic tables only replace concrete field values with symbols, so the number of rows in the table is often fixed. Verifying correctness on a symbolic table of 2 rows alone only guarantees the correctness on any table with 2 rows but insufficient for table of other sizes, and it would be impossible to verify on all sizes up to infinity. Prior work overcomes this by either limiting the scope of query verified to only SPJ (selection, projection and join) queries which bounded verification can be proved to be sufficient for full verification [65, 75], or proposing specific verification mechanism to optimize one type of operator like FGH-rules that optimizes recursive datalog queries [64] or SparkLite that optimizes aggregation operator [37]). However, the scope of queries is either a subset of or different from the data science operators, and their techniques cannot be applied for predicate pushdown optimization.

We propose a more general method to tackle this limitation for predicate pushdown on a set of operators. Specifically, we introduce **pre-conditions** for **small model property (SMP)** [54], which are sufficient to extend a bounded verification to a full verification on table of all sizes. They essentially enables an inductive proof in which as long as the correctness holds on a table of $\leq N$ rows, it holds on a table of $N + 1$ rows. The pre-conditions satisfy SMP itself, as a result, **we only need to verify the precondition and the pushdown correctness on the base case, e.g., symbolic table with ≤ 2 rows, and naturally obtain a full correctness guarantee on a table of arbitrary rows.**

With this novel verification mechanism, we propose MagicPush, a search-verification predicate pushdown optimizer: given an operator with embedded UDF and a predicate on its output, MagicPush enumerates candidate predicates on the operator's input (which may not necessarily be the same as the predicate on the output), and verifies the correctness of each candidate. The goal of

MagicPush is to find predicates on the input table in order to load as little data into the pipeline as possible without changing the pipeline output. To achieve this goal, MagicPush includes a search algorithm to quickly find a selective and correct candidate. It also includes several other techniques to better optimize more pipelines, like 1) searching for additional predicate on the pipeline input when no predicate can be pushed down, 2) handling black-box library functions in UDF, 3) handling NULL values, etc.

In sum, this work makes the following contributions.

- We propose a verification-based method to decide the correctness of predicate pushdown that is able to achieve great coverage of operators, UDFs, and predicates (predicate can contain UDF as well). In particular, we propose pre-conditions for the small-model property that enable efficient correctness checking on bounded-size symbolic tables while providing a full correctness guarantee.
- We explore handling rich library functions by treating them as black-box and leveraging uninterpretable functions in SMT solver while still ensuring pushdown correctness.
- We design a search algorithm to search for the predicate candidates on the input table. It works together with the verification to find the most selective pushed-down predicate, which is often not exactly the same as the predicate to be pushed down.
- We improve the symbolic execution of operators to incorporate the computation of NULL values widely used in data pipelines.
- We extend the process to also return additional predicates on the pipeline input when no predicate can be pushed down. These additional predicates, although doing extra work, reduce the amount of data loaded to the pipeline and often improve performance too.
- We evaluate MagicPush on relational queries (TPC-H) and sampled real-world data science pipelines from GitHub Notebooks. Our results show that MagicPush has high coverage on operators (92.9%) and UDFs (99.7%). Compared to a rule-based baseline that uses a union of rules from prior work, MagicPush is able to discover all pushdown opportunities that are discovered by the baseline in both workloads, and generates more selectively predicates in 42 out of 200 pipelines, reducing the running time by up to 99%.

The paper is structured as follows. We first motivate the challenge with examples in Section 2, followed by introducing the syntax of operators and UDFs that MagicPush supports in Section 3. We present the major technique, how MagicPush pushes predicate through a single operator in Section 4 and how it pushes through a pipeline in Section 5. Then we discuss MagicPush's limitation in Section 6, and finally show the evaluation in Section 7.

2 MOTIVATION

2.1 Limitation of rule-based approach

We use examples abridged from real-world data pipelines to show the challenges of deciding predicate pushdown for non-relational operators and UDFs and why existing rule-based methods would not suffice. We formally define the predicate pushdown problem as follow: **given an operator** Op **and a predicate** F **on its output, we want to search for predicate** G **on its input such that** $F(Op(T)) = Op(G(T))$ **holds on any table** T. We start with individual operators and will discuss predicate pushdown in data science pipelines in Section 5.

Case study 1. Figure 1(1) shows a group-by operator implemented with Pandas API [7] with a max aggregation on the daily revenue. The result of group-by goes through a filter F selecting rows with `max_revenue>1000`. This predicate can be pushed down given the rule described in [44] which says that pushdown is correct under max aggregation when predicates on the aggregation column is in the form of `col_aggr>const` or `col_aggr≥const`.

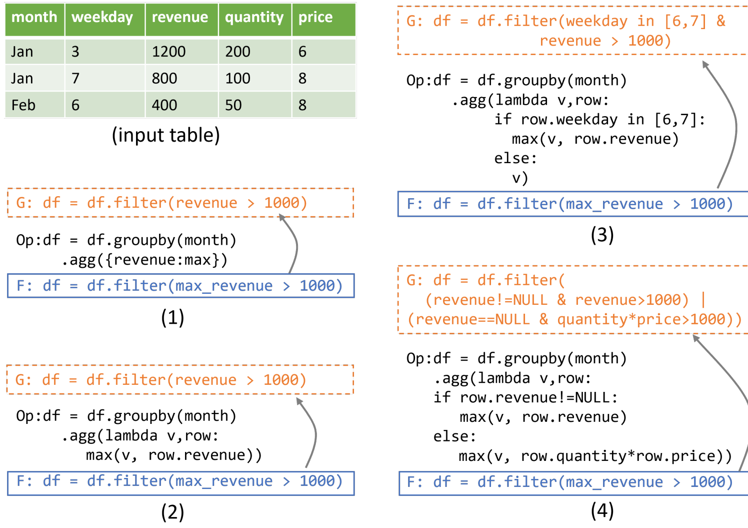


Fig. 1. Pushdown with group-by on UD-aggregations.

Figure 1(2) shows the exact same group-by with a different syntax, where the `max` aggregation is defined as a UDF. This UDF works like a reduce function that takes an accumulative value v (initially 0, omitted for simple illustration) and a row, applying accumulatively to each row in the group. The above rule to pushdown through max aggregation should extend to analyze the UDF syntax to understand that it is doing a `max` aggregation.

Figure 1(3) shows a variation of this group-by operator, where it counts the max weekend revenue. Although the exact predicate can be pushed down, a more selective predicate G, `weekday in [6,7] & revenue>1000`, is also correct. A new rule is needed to generate this more selective predicate that checks `if` condition in UDF.

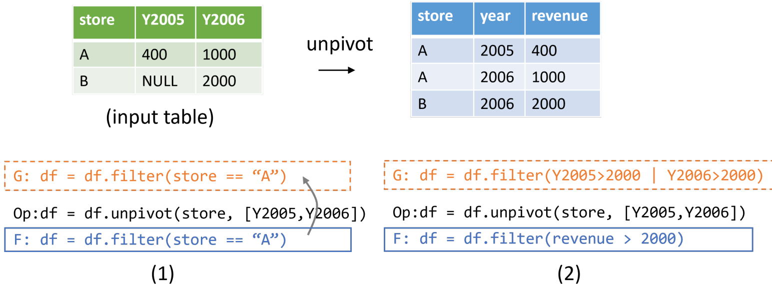
However, not all `if`-conditions can be directly added to G. Figure 1(4) shows an aggregation taking `price*quantity` as revenue when the `revenue` field is `NULL`. This branch condition, `revenue!=NULL & revenue>1000`, cannot be directly pushed down as in (3). Instead, a correct G should be `(revenue==NULL & price*quantity>1000) | (revenue!=NULL & revenue>1000)`, which requires careful rewrite to combine F and the predicate extracted from the `if`-condition.

From this case study, we can see that even though the pushdown for max aggregation has been proposed before, it is far from enough for various max-like user-defined aggregations.

Case study 2. Figure 2 shows an unpivot computation, expanding each row of multiple years' revenue (column `Y2005` and `Y2006`) to a set of rows, each containing one year and its corresponding revenue. Pushdown rules discussed in earlier work [31] pushes a predicate on the `store` column down to the unpivot input, as illustrated in Figure 2(1). However, predicate on other columns, like the `revenue` column in Figure 2(2) (`revenue>2000`) have not been discussed before. Such predicate can be added to reduce the amount of data processed by pivot (keeping F instead of simply pushing F down), where G looks very different from F.

Pushdown rules for pivot and unpivot have been studied by prior work, yet not comprehensive enough to cover widely-exist opportunities. Besides, other non-relational operators like `get_dummies` [9] and `explode` [8] have not been studied and no rule is proposed.

Case study 3. Figure 3 shows a UDF embedded in Pandas `apply` API [5]. This UDF adds a new column `relative_rev` to table `df` computing the relative revenue compared to the same store's revenue

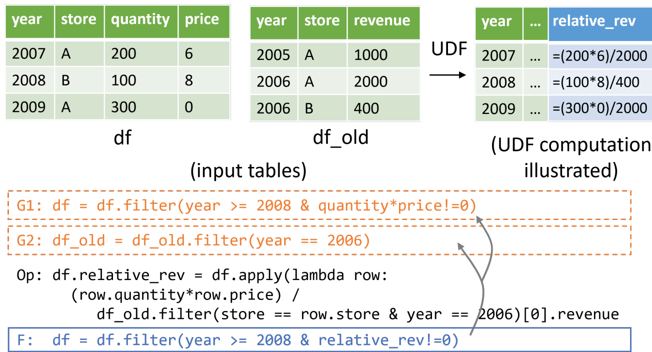


store	Y2005	Y2006
A	400	1000
B	NULL	2000

store	year	revenue
A	2005	400
A	2006	1000
B	2006	2000

Fig. 2. Predicate pushdown with unpivot operator.

of year 2006, which is obtained from another table `df_old`. To do so, for each row in table `df`, it divides the current row's revenue (i.e., `row.quantity*row.price`) by the first revenue of the same store of year 2006 selected from `df_old`.



year	store	quantity	price
2007	A	200	6
2008	B	100	8
2009	A	300	0

year	store	revenue
2005	A	1000
2006	A	2000
2006	B	400

year	...	relative_rev
2007	...	=(200*6)/2000
2008	...	=(100*8)/400
2009	...	=(300*0)/2000

Fig. 3. Pushdown with a UDF involving two tables.

Assuming a predicate F, `year>=2008 & relative_rev!=0`, on the UDF's output. By looking at the UDF, we know that only rows from `df_old` of year 2006 are used to compute the relative revenue, so a predicate extracted from the UDF, `year==2006` can be added to `df_old`. Meanwhile, F can only be pushed to table `df`, with part of F `year>=2008` directly pushed down; the other part, `relative_rev!=0` involving the newly computed column to remove "dirty" data of 0 relative revenue, can filter earlier on `df`, removing rows with 0 quantity or price that computes 0 revenue. No existing rule is able to handle pushdown for such UDF+predicate and designing new rules is apparently complicated to cover many UDF patterns.

2.2 Limitation of bounded verification

We use examples to show the limitation of bounded verification and why it may lead to incorrect pushdown. Briefly, bounded verification computes $F(Op(T))$ and $Op(G(T))$ on a table T whose cell values are symbols, obtains expressions containing symbols as output, and uses SMT solver [20] to check the equivalence of these two expressions under all possible values of symbols. It is bounded as the number of rows in T is fixed. Although we may verify on multiple Ts with different sizes, it is only reasonable to try Ts with a few rows under time budget, and a wrong choice of the bound would make the verification unsound, as we will show below.

Row-to-row transform operator. We start with a simple operator that transforms a row into another row. Because each row is processed individually, verifying on a single-row T is enough to guarantee the correctness of any table.

Groupby-sum **operator.** For this operator, if we still verify on a single-row T, even when the verification passes on a certain G, i.e., $F(Op(T)) = Op(G(T))$ holds on all single-row tables, G can still be incorrect. Consider Op, G, and F as shown in Listing 1, where the equivalence holds on all single-row tables because the sum of a group is the simply the `revenue` of that row. However, this pushdown is apparently incorrect. In fact, it requires verifying on T with at least two symbolic rows to ensure correctness.

Listing 1. Pushdown through *Groupby-sum*, need ≥2 rows

```
df = df.filter(revenue > 1000) # G
df = df.groupby(month).agg({revenue:sum}) # Op
df = df.filter(max_revenue > 1000) # F
```

Top3 **operator.** Unfortunately, 2 rows may not be enough for other operators. Consider a `Top3` operator with corresponding G and F shown in Listing 2. This G would make $F(Op(T)) = Op(G(T))$ hold on any table of 2 rows as *Top3* has no effect (hence a non-op) on such table. However, this is an incorrect pushdown: assuming a table with 4 rows valued `(0)(1)(2)(3)`, $F(Op(T))$ returns an empty table while $Op(G(T))$ returns `(3)`. In fact, it requires at least 4 symbolic rows to check whether G is correct.

Listing 2. Pushdown through *Top3*, need ≥4 rows

```
df = df.filter(v > 2) # G
df = df.sort_by(v, 'asc').top(3) # Op
df = df.filter(v > 2) # F
```

As we can see, no single universal bound exists for all operators, and this bound can depend on the operator and the predicate. Therefore overcoming this limitation and avoiding case-by-case analysis is crucial to building an automatic pushdown optimizer.

3 OPERATOR AND UDF SYNTAX

Data science pipeline. A data science pipeline consists of processing steps transforming raw data into forms ready to answer business questions or train ML models. We focus on pipelines in which each step is presented as an operator like join or groupby, and operators are connected by dataflow. Such pipelines are quite common in data preparation tools like Trifacta [18], PowerQuery [12], Tableu [16] and data prep libraries like Pandas [7]. A relational query in its logical query plan is also a pipeline composed of operators.

Operator. We mainly focus on Pandas operators that update an existing table or transform table(s) into another table. Pandas has about 145 such operators[1], which are still constantly evolving (as the Pandas APIs are under active development), hence deciding pushdown for each individual operator would become very hard to maintainable. A careful analysis of the APIs reveals that many share similar functionality and how predicate can be pushed through, so we group them into 14 core operators, as listed in Table 1, which simplifies our reasoning and avoids duplicating implementations for pushdown verification.

These 14 operators include three categories. The first are commonly-seen operators like `Filter`, `InnerJoin`, `GroupBy`, etc. They often have specific semantics on how to handle the table and only embed UDFs for flexible local computations on rows, for instance, user-defined predicate in `Filter` or aggregation in `GroupBy`. The second category may embde UDFs that define how tables are processed.

[1]We do not consider operators that do not return a dataframe, like "`info`" that prints out dataframe statistics.

Table 1. Core Operators implemented by MagicPush.

Operator syntax	embedded UDF	input/output type	Description	SMP pre-condition
Filter	predicate	row→bool	selection in SQL	ROW-TO-ROW
InnerJoin	join-predicate	row_1,row_2 →bool	innerjoin in SQL	ROW-TO-ROW
RowTransform	row-transform	row_1 →row_2	user-defined row/scalar value transform	ROW-TO-ROW
DropColumn	NA	NA	projection in SQL	ROW-TO-ROW
Reorder	compare	row_1,row_2 →bool	all ops with no effect under bag semantics	ROW-TO-ROW
GroupBy	aggregation	row_1,row_2 →row_1	group-by and aggregation via UDF	AGGREGATE
Pivot	aggregation	row_1,row_2 →row_1	pivot[10] operator	AGGREGATE
CumAggr	aggregation	row_1,row_2 →row_1	accumulative aggregate prior rows	AGGREGATE
UnPivot	NA	NA	unpivot operator[10]	ROW-EXPAND
RowExpand	explode row	row_1 → $[row_2]_k$	convert one row into up to k rows via UDF	ROW-EXPAND
TopK	compare	row,row→bool	get the first K rows after sorted	TOPK
LeftOuterJoin	join-predicate	row_1,row_2 →bool	left/right outer join in SQL	LEFTOUTERJ
GroupedMap	(df)*-transform	$(table_i)^*$→table	transform grouped sub-tables to sub-table	SUBPIPELINE
RowIterPipeline	(row+(df)*)-transform	row_1,$(table_i)^*$→row_2	iterate rows in a table to output a new row	ROW-ITER-PIPELINE
Other	NA	NA	other operators without correct pushdown	NA

We call such UDF "subpipeline", which resembles a subquery in SQL. This category includes two operators, *GroupedMap* and *RowIterPipeline*: *GroupedMap* embeds UDF specifying how each grouped sub-table is transformed, and *RowIterPipeline* embeds UDF specifying how each row in the left table computes with the right table into a new row (e.g., an UDF shown in Figure 3 contains a subpipeline composed of operators *Filter*, *RowTransform* and *Groupby* as described in Figure 8). The third category noted as "Other" in Table 1 represents operators where predicates cannot be pushed through, like `transpose`, and we will discuss them in Section 6.

UDF embedded in operator. Many Pandas APIs accept function as a parameter, like `apply`, `transform`, `query`, etc. MagicPush supports such UDFs embedded in an operator with specific input-output type as listed in Table 1, e.g., InnerJoin with custom join predicate which takes in two rows with different schemas and returns a boolean.

```
op           :=  Filter(table, UDF)
                 InnerJoin(table, table, UDF)
                 RowTransform(table, UDF)
                 DropColumn(table, UDF)
                 Reorder(table, UDF)
                 GroupBy(table, [column]+, UDF)
                 Pivot(table, ([column]+), UDF)
                 CumAggr(table, UDF)
                 Unpivot(table, [column]+, UDF)
                 RowExpand(table, UDF)
                 TopK(table, UDF)
                 LeftOuterJoin(table, table, UDF)
                 GroupedMap(table, [column]+, UDF)
                 RowIterPipeline(table, table, UDF)
UDF          :=  ('lambda' [r-param]+ : [stmt]+ return_stmt) | ('lambda' [t-param]+ : [op]+)
r-param      :=  scalar | row | [row]_k
scalar       :=  int | string | float | bool | ...
row          :=  [scalar]_k
t-param      :=  table | row
table        :=  [row]_n
stmt         :=  assignment | if_stmt | for_stmt | while_stmt | import_stmt |
                 with_stmt | del_stmt | 'pass' | 'break' | 'continue'
return_stmt  :=  'return' r-param
op           :=  Filter(table,UDF) | InnerJoin(table,table,UDF) | ...
```

The syntax of operators and UDF is listed as above. The first category of operator embeds UDF accepting parameters with a constant size like scalar value, row or row set with a constant size (i.e., `r-param`) and returning a constant-size `r-param`. MagicPush supports UDF in Python syntax, consisting of core Python statement types [15] like assignment, `if` and `for` statement (i.e., `stmt`), only excluding statements that might have side effects like `assertion` or `raise` that throws error. To achieve this, MagicPush includes a Python analyzer to analyze input and output types of UDF, as well as a Python interpreter that can compile and execute Python code symbolically during verification.

The second operator category, *GroupedMap* and *RowIterPipeline*, embed UDF that may receive tables (i.e., `t-param`) as parameter. The body of such UDF is a subpipeline composed of other operators (i.e., `op`), which has the same syntax as in a normal pipeline.

Comparing with SQL UDFs. UDFs in data science pipelines slightly differ from SQL UDFs in two ways. First, data pipeline UDFs widely use Python libraries compared to a much smaller set of functions in SQL UDFs, hence supporting uninterpreted libraries is crucial for MagicPush. Second, data pipeline UDFs are mostly embedded in APIs due to the rich and flexible API design while SQL UDFs sometimes contain cursor loops that actually perform an operator's work. For example, an aggregation that conditionally concatenates a string column into one string may be implemented with a cursor loop because some DMBS may not support such aggregation function, while Pandas' `groupby` API can embed any aggregation written in Python. So we focus on supporting embedded UDFs but not cursor loops. We believe prior work like Aggify[39], QBS[26] and Casper[22] can be leveraged to rewrite loop into operator and MagicPush can optimize for SQL UDFs as well.

4 PUSHDOWN FOR SINGLE OPERATOR

We first introduce how MagicPush pushes a predicate through one operator, i.e., given F and Op, find the most selective predicate G such that $F(Op(T)) = Op(G(T))$. We also consider predicate G that additionally adds to the input without removing F when F cannot be pushed down (i.e., $F(Op(T)) = F(Op(G(T)))$), which may still improve performance by making Op process less data. For illustration purposes, we call the former case equivalent pushdown and the latter superset pushdown.

Figure 4 shows the workflow. The predicate search module returns one candidate G each time following selectivity order (more selective first). Each candidate G then goes through equivalence verification to check whether $F(Op(T)) = Op(G(T))$. Upon passing, MagicPush returns this G and stops. Otherwise, it moves on to superset verification and returns G if passes. When both fail, MagicPush generates the next candidate. Because MagicPush verifies more selective candidate prior to less selective one, it always returns the most selective G that is correct, either equivalent or superset. MagicPush tries a limited number of candidates and returns a `true` filter (that filters out no rows) if no correct filter is found.

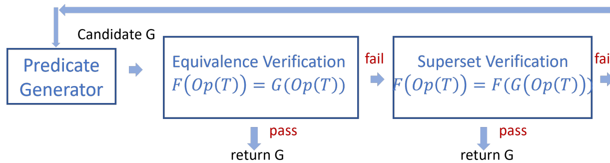


Fig. 4. How MagicPush searches for G given Op and F.

When verifying a candidate, shown in Figure 5, MagicPush creates a symbolic table based on the input table schema. It runs symbolic execution on this symbolic table to check whether F, Op, and G satisfy the pre-condition for the small-model property. If not, MagicPush would not be able

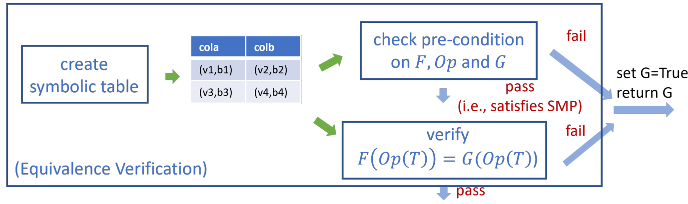


Fig. 5. How MagicPush verifies one candidate G.

to guarantee G's correctness and simply decides that G fails. Then MagicPush continues to check the verification target $F(Op(T)) = Op(G(T))$ on the symbolic table and returns fail or pass. The process for superset verification is similar, only to replace the pre-condition and the verification target.

4.1 Bounded verification with SMT solver

In this section we introduce how to verify $F(Op(T)) = Op(G(T))$ using symbolic execution. MagicPush computes this equation on a symbolic table of fixed size, producing a logic formula to be solved using the SMT solver. This technique has been used in prior work [27, 56, 62, 66] and we adapt it for data science operators. Specifically, we introduce two extra symbols, a *NULL indicator* associated with each cell value and a *row-exist indicator* associated with each row, and modify the symbolic execution to incorporate the computing of these two symbols.

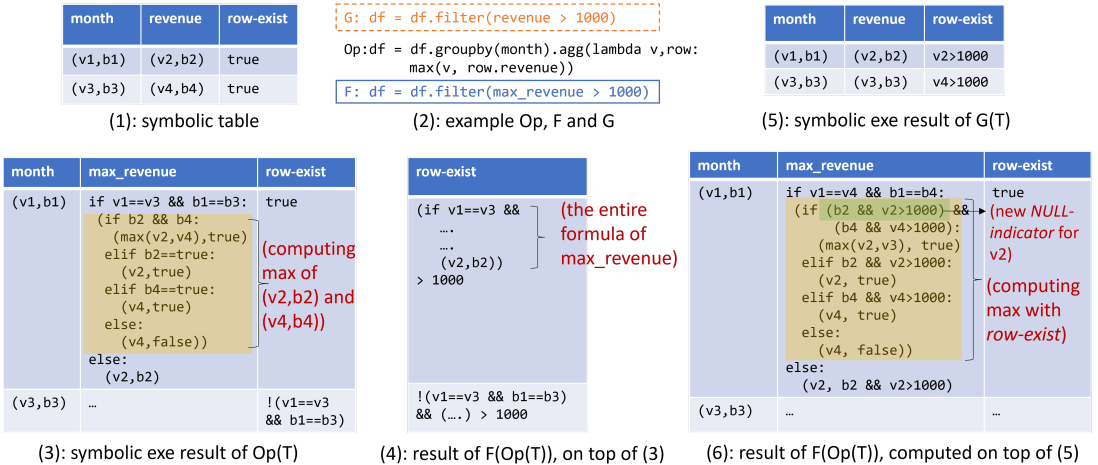


Fig. 6. Demonstration of symbolic execution with *NULL indicator* and *row-exist indicator*.

Symbolic table representation. We use symbols to represent values in a table rather than concrete values. To handle NULL values, we add a NULL-indicator, i.e., a symbol attached to each value to represent whether the value is not null, hence each cell is represented by a pair of symbols. An example shown in Figure 6(1), where symbols like `v1,v2` represent column values and `b1,b2` are NULL-indicators (`b1` is true when `v1` is *not* null, etc). We add another symbol associated with each row, `row-exist`, an indicator where `true` value means that the row exists. This indicator is `true` in the initial symbolic table and will be computed after applying filter F or G.

Running filters. As filters are running on a symbolic table instead of a concrete table, whether a row survives a filter is computed from symbols of that row. As we introduce additional `row-exist`

symbol, running filter simply resets this symbol, as an example shown in Figure 6(5): after running filter G, `revenue>1000`, the `row-exist` symbol is updated to an expression like `v2>1000` where `v2` is the symbol representing the first row's revenue.

Symbolic operator execution. We build a Python interpreter with symbolic execution that allows running arbitrary UDF in Python syntax and implement each operator in Python. Different from standard symbolic program execution [60], MagicPush computes NULL-indicator associated with each value. Specifically, at each computational byte-code like binary add, MagicPush returns a pair of values: an `if-else` expression choosing from four potential results depending on the values of two inputs are NULL or not, and an expression as the result's NULL-indicator. For instance, the yellow-shaded area in Figure 6(3) shows the computation of a binary `max`: MagicPush returns a pair of expressions, the max of values `v2,v4` when both are not null and `true` as NULL-indicator, or `v2` when only `v2` is not null and `true` as NULL-indicator, etc.

The `row-exist` symbol may affect NULL-indicator because rows in the symbolic table may not be present, so the actual NULL-indicator is the conjunction of both its original NULL-indicator and corresponding `row-exist` value. For instance, in Figure 6(6), when computing the `max`, the actual NULL-indicator used for revenue `v2` becomes `b2 && v2>1000`, and similarly for `v4`.

Checking $F(Op(T)) = Op(G(T))$. The equivalence checking examines pair of values in every cell in the output symbolic table after running $F(Op(T))$ and $Op(G(T))$ to see whether they are equivalent using the constraint solver. Due to the presence of NULL-indicator, the two cells are equal only when their NULL-indicators are equal, and their values are equal when the NULL-indicator is true (i.e., their values only matter when not null). Presented formally, two cells, represented by (v, b) and (v', b') are equal when $b == b' \wedge (b \rightarrow v == v')$.

4.2 Extending to full verification

The verification using SMT solver is bounded, and in this section, we introduce how to extend it to a full verification using pre-condition. Briefly, we find pre-condition for small model property (SMP) [46, 47]: as long as Op, F and G satisfy the pre-condition, verification of $F(Op(T)) = Op(G(T))$ on a small, finite-size table yields a full verification. Furthermore, the pre-condition is an expression to be computed from F, Op, and G, and itself also satisfies SMP. With such a pre-condition, we are able to obtain a full verification by merely checking the pre-condition, then $F(Op(T)) = Op(G(T))$ on the small, finite-size symbolic table.

The magic behind such a pre-condition is that it enables inductive proofs of $F(Op(T)) = Op(G(T))$. When the pre-condition is satisfied, and the equivalence holds on all tables T where $|T| \leq N$, we can prove that it holds on all tables T' where $|T'| = N + 1$. With such proof, we only need to show the equivalence on the *base case*, i.e., a table with a small number of rows, and the inductive proof guarantees the correctness of all tables.

These pre-condition varies across operators; we summarize core operators into 6 categories with their pre-conditions and base-case table size listed in Table 2. Due to the limited space, we only sketch the proof for three categories, including the inductive proof for $F(Op(T)) = Op(G(T))$ and a proof showing that the pre-condition itself satisfies SMP. The complete proof is implemented in Coq [1] (an interactive proof assistant) and released at [13]. Superset verification ($F(Op(T)) = F(Op(G(T)))$) is similar only with different pre-conditions, which are listed in Table 2.

4.2.1 Row-to-row transformation. Pre-condition: None

Row-to-row transformation requires no pre-condition because it processes each row individually. This is the simplest case and has been discussed in prior work [46, 47]. We use it to provide a context to help understand the proofs of other operators. We break T' into table T_1 of N rows and another table with only one row r_{N+1} (i.e., $T' = T_1 + \{r_{N+1}\}$ where + is table union), hence:

$$\begin{aligned} F(Op(T')) &= F(Op(T_1) + Op(\{r_{N+1}\})) && \text{row processed individually} \\ &= F(Op(T_1)) + F(Op(\{r_{N+1}\}) && \text{filter processed individually} \\ &= Op(G(T_1)) + Op(G(\{r_{N+1}\})) && \text{inductive assumption} \\ &= Op(G(T_1 + \{r_{N+1}\})) = Op(G(T')) && \text{filter processed individually} \end{aligned} \tag{1}$$

Therefore we have shown that the equivalence $F(Op(T')) = Op(G(T'))$ holds on any table T' of size $N + 1$.

4.2.2 Aggregate. Pre-condition:

1. Aggregation func $Aggr$ is associative and commutative.
2. $\forall T, |T| = k, k \geq 2, \exists T_x, |T_x| = |T| - 1, s.t. Aggr(T) = Aggr(T_x)$.
3. $\forall U, V, U \neq \emptyset \wedge V \neq \emptyset \wedge F(Aggr(U + V)) = \emptyset \Leftrightarrow F(Aggr(U)) = \emptyset \wedge F(Aggr(V)) = \emptyset$.
4. $\forall U, V, F(Aggr(U + V)) \neq \emptyset \wedge G(V) = \emptyset \Rightarrow Aggr(U + V) = Aggr(U)$.

where T, T_x, U, V are tables and $Aggr$ is an aggregation on table.

This category involves aggregating a set of rows into a single row. We explain these four conditions intuitively. First, the $Aggr$ function is associative and commutative, meaning the order of rows has no effect on the aggregation result. Second, the for every table with ≥ 2 rows, there always exists a smaller table (whose content may be completely different) that computes the same aggregation result. Third, when a table's aggregation result does not pass F, any of its non-empty subset's aggregation does not pass F either; and reversely, when two table's aggregation both fail F, their union's aggregation will not pass F. Fourth, if a table's aggregation passes F, its subset that fails on G has no effect on the aggregation result.

We first show the inductive proof with goal $F(Aggr(T')) = Aggr(G(T'))$. We discuss three cases: 1) $Aggr(T')$ passes filter F; 2) we split rows in T' into U and V by whether they passes G, and none passes G; 3) some rows in T' passes G. The proof is as follow.

$$\begin{aligned} & \text{if } Aggr(T') \text{ does not pass } F \\ F(Aggr(T')) &= F(Aggr(T_1 + \{r_{N+1}\})) && \text{split } T'\text{: } T' = T_1 + \{r_{N+1}\} \\ &= F(Aggr(T_1)) + F(Aggr(\{r_{N+1}\})) && \text{pre-condition(3), each is } \emptyset \\ &= Aggr(G(T_1)) + Aggr(G(\{r_{N+1}\})) && \text{inductive assumption} \\ &= Aggr(G(T_1 + \{r_{N+1}\})) = Aggr(G(T')) && \text{aggregate } \emptyset\text{s distributive} \\ & \text{if } Aggr(T') \text{ passes } F \text{ and } V \neq \emptyset \\ F(Aggr(T')) &= F(Aggr(U + V)) = F(Aggr(U)) && \text{pre-condition(4)} \\ &= Aggr(G(U)) && \text{inductive assumption} \\ &= Aggr(G(U + V)) = Aggr(G(T')) && G(V) = \emptyset \\ & \text{if } Aggr(T') \text{ passes } F \text{ and } V == \emptyset \\ F(Aggr(T')) &= Aggr(T') && \text{case condition} \\ &= Aggr(G(T')) && \text{case condition} \end{aligned} \tag{2}$$

Furthermore, these pre-conditions should hold on any-size tables. Next, we prove that they satisfy SMP themselves. We similarly show an inductive proof, where for each pre-condition, assume it holds on all tables of size $\leq N$ and proves it holds on T' of size $N + 1$.

We skip pre-condition(1) as associativity and commutativity satisfy SMP trivially. The proof for pre-condition(2)(3)(4) is shown in Equation(3)(4)(5), respectively. We present the main idea behind the proof and skip corner cases (e.g., when $|U| \leq 2$) due to limited space. We provide the complete proof in [13].

$$
\begin{aligned}
Aggr(T') &= Aggr(T_1) \circ Aggr(\{r_{N+1}\}) && \text{split } T'\colon T' = T_1 + \{r_{N+1}\} \\
&= Aggr(T_{1x}) \circ Aggr(\{r_{N+1}\}) && |T_{1x}| = |T_1| - 1, \text{ inductive assumption} \\
&= Aggr(T_{1x} + \{r_{N+1}\}) && \text{set } T'_x = T_{1x} + \{r_{N+1}\}, |T'_x| = |T'| - 1
\end{aligned} \tag{3}
$$

$$
\begin{aligned}
&\text{direction} \Rightarrow, \text{assume } F(Aggr(U+V)) = \emptyset \\
&\Rightarrow \exists U_x, |U_x| = |U| - 1 \wedge Aggr(U) = Aggr(U_x) && \text{pre-condition(2)} \\
&\Rightarrow F(Aggr(U_x + V)) = \emptyset && \text{assumption} \\
&\Rightarrow F(Aggr(U_x)) = \emptyset \wedge F(Aggr(V)) = \emptyset && \text{inductive assumption} \\
&\Rightarrow F(Aggr(U)) = \emptyset \wedge F(Aggr(V)) = \emptyset && \text{definition of } U_x \\
&\text{direction} \Leftarrow, \text{assume } F(Aggr(U)) = F(Aggr(V)) = \emptyset \\
&\Rightarrow F(Aggr(U)) = F(Aggr(U_x)) = \emptyset && \text{assumption} \\
&\Rightarrow F(Aggr(U_x + V)) = \emptyset && \text{inductive assumption} \\
&\Rightarrow F(Aggr(U + V)) = \emptyset && \text{definition of } U_x
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
&\text{assume } F(Aggr(U+V)) \neq \emptyset \wedge G(V) = \emptyset \\
&\Rightarrow \exists U_x, |U_x| = |U| - 1 \wedge Aggr(U) = Aggr(U_x) && \text{pre-condition(2)} \\
&\Rightarrow F(Aggr(U+V)) = F(Aggr(U_x + V)) \neq \emptyset && \text{assumption, definition of } U_x \\
&\Rightarrow Aggr(U_x) = Aggr(U_x + V) && \text{induction assumption} \\
&\Rightarrow Aggr(U) = Aggr(U + V) && \text{definition of } U_x
\end{aligned} \tag{5}
$$

4.2.3 RowIterPipeline. Pre-condition:

(1) $\forall$ row r_l which is the input to the UDF, each operator in the subpipeline in UDF satisfies SMP when treating r_l as a constant value.

(2) $\forall$ row r_l, table T_{r_i} which are the UDF's input, $G_l(\{r_l\}) = \emptyset \Rightarrow F(Op(\{r_l\}, T_{r_1}, ...)) = \emptyset$.

(3) $\forall$ row r_l, table T_{r_i} which are the UDF's input, $G_l(\{r_l\}) = \{r_l\} \Rightarrow F(Op(\{r_l\}, T_{r_1}, ...)) = Op(\{r_l\}, G_1(T_{r_1}), ...)$.

This category represents a type of UDF that takes each row of one table (we call it the left table) and some other tables (the right tables) as input and outputs a new row. We assume the computation inside the UDF can be represented as a subpipeline, with an example shown in Figure 3. Intuitively, the first pre-condition says that when treating the row from the left table as a constant, the UDF becomes a subpipeline that takes only the right tables as input, and each operator in the subpipeline satisfies SMP. The second pre-condition says that, if a row in the left table r_l does not pass its filter G_l, the output of r_l with any right table does not pass F. The third pre-condition deals with the case when r_l passes G_l, then treating r_l as a constant value, the pushdown of the predicate on the right table on the subpipeline is correct.

We first show the inductive proof with goal $F(Op(T'_l, T'_{r_1}, ...)) = Op(G_l(T'_l), G_1(T'_{r_1}), ...)$. Because each row in the left table is processed individually, following the reasoning of row-to-row transformation, as long as we show proof for the table of size 1 for the left table, the proof holds for T_l of any size. We mainly focus on the proof of the right table (assuming only 1 table for simple illustration), and discuss two cases when the single-row r_l in the left table passes/fails filter G_l.

$$
\begin{aligned}
&\text{if } r_l \text{ passes } G_l \\
F(Op(\{r_l\}, T'_r)) &= Op(\{r_l\}, G_r(T'_r)) && \text{pre-condition(3)} \\
&= Op(G_l(\{r_l\}), G_r(T'_r)) && \text{assumption } r_l \text{ passes } G_l \\
&\text{if } r_l \text{ does not pass } G_l \\
F(Op(\{r_l\}, T'_r)) &= \emptyset && \text{pre-condition(2)} \\
&= Op(G_l(\{r_l\}), G_r(T'_r)) && Op \text{ returns } \emptyset \text{ as no row input provided}
\end{aligned} \tag{6}
$$

Next, we show that these pre-conditions satisfy SMP themselves. The first pre-condition is trivial as it is checking SMP itself. For (2)(3), as each operator in a subpipeline satisfies SMP, the pushdown

Table 2. Pre-conditions for each category (including equivalence and superset), base-case table size, and whether they allow black-box functions. "Category" corresponds to operators as shown in Table 1.

Category	Embedded UDF	Allow black-box function in UDF?	Equivalence pre-condition	Superset pre-condition	Base-case table size n
ROW-TO-ROW	row transform	Yes	None	None	1
AGGREGATE	aggregation function	Yes if predicate includes only group-by cols	As listed in Section 4.2.2	Equiv pre-condition (1)(2)(4) (3)$\forall U, V$, $F(Aggr(U+V)) = \emptyset \wedge$ $G(V) = \emptyset$ $\Rightarrow F(Aggr(U)) = \emptyset$.	2, # columns of output table for pivot
ROW-EXPAND	convert one row into K rows	Yes	None	None	1
TOPK	compare function	equivalence:No superset:Yes	$\forall T, \vert F(topK(T))\vert < K \Rightarrow$ $(\forall r, r \leq min_{UDF}(T)$ $\Rightarrow G(\{r\}) = \emptyset)$.	None	K+1
LEFTOUTERJ	join predicate	Yes	(1)$\forall t_l, t_r, T_r$, $F(LJ(\{t_l\}, \{t_r\})) = \emptyset$, $pred_j(t_l, t_r) =$ True $\Rightarrow F(LJ(\{t_l\}, T_r)) = \emptyset$ (2)$\forall t_l, t_r, F(LJ(\{t_l\}, \{t_r\})) \neq \emptyset$, $pred_j(t_l, t_r) =$ True $\Rightarrow G_r(\{t_r\} \neq \emptyset$	Equivalence pre-condition(2)	2
SUBPIPELINE	(df)*-transform	Yes if all subpipeline op allows BB func	Each operator in subpipeline satisfies SMP.	same as equivalence pre-condition	$max(n)$ of subpipeline operators
ROW-ITER-PIPELINE	(row+(df)*)-transform	Yes if all subpipeline op allows BB func	As listed in Section 4.2.3	same as equivalence pre-condition	$max(n)$ of subpipeline operators

of the entire subpipeline also satisfies SMP because the process of pushdown for each operator is independent.

4.3 Handling uninterpretable functions

Library functions are widely used in pipeline UDFs, particularly UDFs written in Python in which rich libraries are available. It would be a huge work to interpret each library function in verification. Fortunately, sometimes a library function can go through verification without actually computing it. For instance, an *RowTransform* operator that assigns a new column `day` to a table computing `dayofweek` from an original column `date` (`df.day=dayofweek(df.date)`), and an F on its output that selects weekends (`df.day in [6,7]`). MagicPush will generate a candidate G `dayofweek(date) in [6,7]`. Leveraging uninterpreted function [19] in the SMT solver, MagicPush is able to verify G's correctness presented by the expression `(dayofweek(date)) in [6,7]==dayofweek(date) in [6,7]` (the left-hand side is the computed `row-exist` symbol of $F(Op(T))$ and the right-hand side $Op(G(T))$) without unboxing `dayofweek`.

However, not all black-box functions can remain unboxed and may need to be computed to verify for some operators. For example, pre-condition(4) of group-by requires checking $Aggr(U+V) = Aggr(U)$. We assume U and V are single-row tables with one column represented by v_u and v_v for ease of explanation, and f is a black-box aggregation function. Verifying this pre-condition checks expression $f(f(v_{init}, v_u), v_v) = f(v_{init}, v_u)$, where the left-hand side calls f twice while the right-hand side calls f once. This expression cannot be verified unless f is computed, and MagicPush will return "fail" in this case. Although whether a black-box function is allowed depends on the verification, we summarize when it is empirically allowed in Table 2.

4.4 Predicate search

We now introduce how to search for candidate predicate G. Because of the cost to verify each G, we cannot afford to try all possible candidates. Instead, MagicPush enumerates candidates following their selectivity and stops when it finds a correct one. Doing so would return the most selective G without verifying all candidates.

The challenge is to have good coverage such that the best correct G is included and enumerated as early as possible without wasting verifying too many incorrect candidates. MagicPush obtains predicate components from F and Op, then synthesizes candidate G by composing these components using conjunction and disjunction. We do not assume knowing predicate cardinality, so MagicPush approximates the selectivity order using the number of conjunctions and literals (i.e., comparisons) in conjunction.

4.4.1 Extracting predicate components. MagicPush first extracts components from F. As operators can change table schema (e.g., changing column type, renaming columns, pivot table, etc.), F usually cannot be directly pushed to the input table as it may contain newly produced columns that do not exist in the input tables. These new columns are usually computed from some columns in the input table, so carefully replacing them with corresponding input columns can often lead to a legitimate candidate G. Therefore MagicPush builds a mapping between output columns and related input columns based on the operator semantics, and replaces each newly-generated column c' in F according to the mapping:

- c' directly maps to exactly one input table column c, like in column-renaming operator, then replace c' with c.
- c' maps to input columns by an expression, e.g., setting a new column by an UDF like (`df.revenue=df.price*df.quantity`, then replace c' with the expression. If the UDF contains other variables than table fields and constants (e.g., group-by's accumulative variable `v` in Figure 1(2)(3)(4)), MagicPush extracts sub-expressions that compute over only table fields and constants (e.g., `row.revenue` and `row.quantity*row.price` in Figure 1(4) but not `max(v, row.revenue)`), and produces a set of components each replacing c' with one expression.
- c' maps to no columns, like *DropColumn* operator, then remove any comparison or sub-expression in F containing c'. Removing comparison is done by replacing that comparison with `true`; if no expression/comparison except `true` is left after removing, MagicPush adds a `false` predicate as component.
- c' maps to a set of columns $c_1, c_2, ...$, e.g., unpivot shown in Figure 2(2) where the `revenue` column maps to column `2005` and `2006` in the input table), then MagicPush replaces c' with c_i and generates two components by combining all replaced expressions using disjunction or conjunction.

MagicPush then extracts components from the operator itself as it may contain predicates that can also be pushed to G. For instance, the branch condition selecting weekends in Figure 1(3) can be pushed down to the input. MagicPush extracts every branch condition in the UDF (discarding comparisons that contain anything other than table fields and constants), and uses each branch condition as well as its negation (e.g., both `row.weekday in [6,7]` and `!(row.weekday in [6,7])`) as two components. It also extracts predicates from the operator if it is a *Filter* operator as a component. The above process can produce a set of components, and next, we show how MagicPush combines them into a candidate G.

4.4.2 Synthesizing candidate G. The algorithm to combine components to produce candidate G is shown in Algorithm 1. After extracting components (Line 1-2), MagicPush first returns a "shortcut": a candidate that we observe frequently to be the most selective and correct G (Line 4-7). This shortcut is computed as follows: if we cannot extract components from Op, then we return the

Algorithm 1 Combining Predicate Components

```
1: procedure CandidateGen(F,Op)
2:     comps1 ← extractFromF(F,Op)
3:     comps2 ← extractFromOp(F,Op)
4:     if len(comps2) == 0 then
5:         yield comps1[0]
6:     else
7:         yield comps1[0] ∧ (∧(extractNonConflict(comps2)))
8:     components ← comps1 ⋃ comps2
9:     conjunctions,candidates ← ∅
10:    for subset_p ∈ subset of conjunctions do
11:        if not hasConflictLiteral(subset_p) then
12:            conjunctions.add(∧(subset_p))
13:    sort conjunctions by the number of literals in desc order
14:    for subset_j ∈ subset of conjunctions do
15:        if not duplicated(candidates,subset_j) then
16:            candidates.add(∨ subset_j)
17:            yield ∨(subset_j)
```

rewritten F (by replacing newly-generated column c' as described in Section 4.4.1) alone as a candidate (Line 5); otherwise, return a conjunction of the rewritten F and components extracted from Op (Line 7). When conjuncting, we take only a branch condition l when both l and $\neg l$ are included to avoid conflicting literals.

Then MagicPush falls back to the enumeration procedure after the shortcut. It first enumerates all possible conjunctions of literals from components (Line 10-12). For each subset of components, if it does not contain conflicts (e.g., l and $\neg l$), then MagicPush conjuncts components in the subset and adds it to `conjunctions` list. Then MagicPush sorts the `conjunctions` list by the number of literals (Line 13): doing so allows choosing conjunctions with more literals first in the next loop. After sorting, MagicPush produces candidates by enumerating subsets of `conjunctions` list (Line 14-17) and disjuncting each subset. In this loop, we choose the subset with fewer elements first following the order of the `conjunctions` list. That is, if `conjunctions` has 3 elements $a \wedge b$, a, b, MagicPush returns one-element disjunction $a \wedge b$ first, next a, b, then multi-element disjunction like $a \vee b$.

5 PREDICATE PUSHDOWN FOR PIPELINE

In this section, we introduce how MagicPush pushes any predicate in the middle of a pipeline to its input, particularly when dataflow diverges or is conditionally executed. We also introduce the pushdown procedure for operators containing subpipeline like *GroupedMap* and *RowIterPipeline*.

5.1 Pushing through dataflow

We first look at the basic scenario where each operator has one input and one output, as shown in Figure 7(1). MagicPush decides the pushdown for each individual operator from the last operator to the first along the data flow: it first finds G_2 given the last operator Op_2 and filter F, where G_2 works on df_2, the output table produced by the prior operator Op_1. Then MagicPush pushes G_2 through Op_1 and obtains G_1, the filter on the input table df_1. If any G obtained on any operator is a superset pushdown, then we keep the original filter F in the pipeline, otherwise, F would be removed.

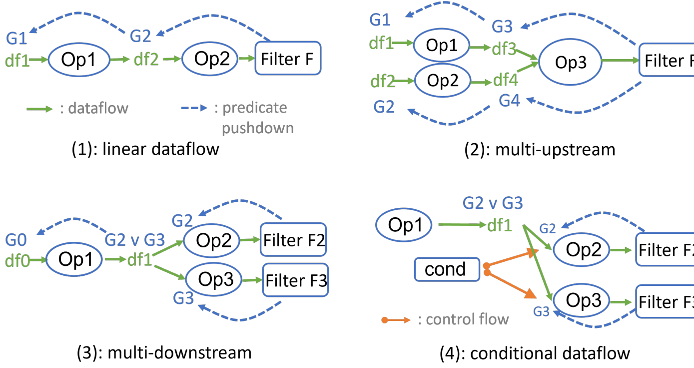


Fig. 7. Handling multi up/down-stream and branches.

When an operator has multiple upstreams like an innerjoin, MagicPush naturally finds the predicate for each table when deciding predicate pushdown as it searches for G_i on each table T_i and verifies $F(Op(T_1, T_2, ...)) = Op(G_1(T_1), G_2(T_2), ...)$.

In the case of multiple-downstream where the output of one operator is used by multiple downstream operators, as shown in Figure 7(3). After deciding pushdown for operator Op_2 and Op_3 and obtaining G_2 and G_3 correspondingly, the output filter of their common upstream operator Op_1 becomes $G_2 \vee G_3$, as the table df_1 should contain the data for both Op_2 and Op_3 after filtering, hence taking disjunction. Then MagicPush pushes $G_2 \vee G_3$ through Op_1. Note that the result pipeline needs to be changed by adding filter G_2 before Op_2 and G_3 before Op_3 because the $df1$ contains rows for both Op_2 and Op_3 and needs to split to send corresponding rows to each downstream operator.

When the pipeline contains conditional branches, as illustrated in Figure 7(4), MagicPush similarly takes the disjunction of predicates pushed from each path (i.e., $G_2 \vee G_3$). This is because each path can be potentially taken, so a table entering both paths should include all data that might be used by either path.

5.2 Pushdown for operators with a subpipeline

Unlike other operators, the search of G for operators with a subpipeline follows the procedure to push the predicate through pipelines operator-by-operator until reaching the subpipeline's input. The procedure is slightly modified for *RowIterPipeline* as it involves pushing down the predicate to the input row instead of the input table. We illustrate using the example from Figure 3, and the procedure is shown in Figure 8. First we split F into two predicates: one involving c' generated by the UDF (e.g., `relative_rev!=0` where `relative_rev` is newly-generated) which is pushed down operator-by-operator till the inputs (e.g., resulting in `quantity*price!=0` on table `df` and `year==2006` on table `df_old`), and the remaining part of F without c' (e.g., `year>=2008`) is pushed directly to the left table. When the two pushed predicate, `quantity*price!=0` and `year>=2008`, merges at table `df`, they are combined using conjunction or disjunction, producing two candidates Gs to be verified. In this example, the conjunction is returned as it passes the verification for this subpipeline.

6 DISCUSSION AND LIMITATION

Unhandled operators. For some operators, predicates cannot be pushed through without additional effort and therefore cannot be handled by MagicPush. These operators belong to three categories. First, if each row in the output relies on all rows from the input, like `transpose` and `corr`

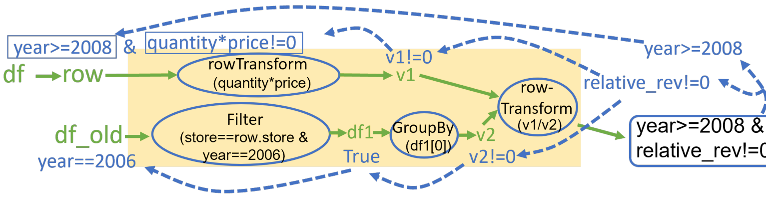


Fig. 8. Predicate pushdown with subpipeline.

(which computes the correlation among columns), predicate filtering out any row in the input will lead to a different output. Second, if the operator is position sensitive, for instance, `shift` and `rolling` in which each output row is computed from its neighboring rows. No predicate can be pushed down as it would be impossible to keep neighboring rows using only predicate but not position index, which usually does not exist in tables. Third, if the operator's output is nondeterministic, like `sample`, the correctness of pushdown can not be easily verified.

Unhandled UDFs. MagicPush is not able to handle a UDF if it 1) contains an uninterpretable function but verification requires computing the function, as discussed in Section 4.3; or 2) has a different input-output type than the one handled by MagicPush (as listed in Table 1), like the `mode` aggregation that takes an additional state as input to keep the count of each distinct value. However, as we will see in the evaluation, such UDFs do not often appear in real-world pipelines built with Pandas API and our coverage is already >99% despite such limitations.

Completeness of MagicPush. The verification of pushdown correctness is sound but incomplete, as theoretically not all correct pushdown necessarily satisfies the small model property. For example, a *GroupBy* with a count aggregation does not meet pre-condition(2) listed in Section 4.2.2 as no smaller table would return the same count as a larger table, but some predicate might be pushed down through it. We work around count-like aggregation by adding a `count` column that stores a duplication factor of that row (which can be 0 or any value >= 1), and rewrite the count aggregation to accumulates the value in the `count` column (i.e., `lambda v,row:v+row['count']`). After the rewrite, pre-condition(2) can be satisfied and the correct pushdown with count aggregation will be returned by MagicPush if exists. This rewrite has been used in prior work [62] to accelerate verification, and we leverage it to avoid missing a pushdown opportunity. Other than the count aggregation, we have not seen nor been able to find any other case that MagicPush returns a false-negative result.

Generalizability. The operators and UDFs handled by MagicPush are driven by our analysis of Pandas, SQL and Spark UDFs, which already achieve very high coverage on real-world pipelines (as we will show in Section 7). We do acknowledge that for new operators and UDFs outside of MagicPush's syntax, new pre-conditions need to be developed and proved, making it hard for MagicPush to generalize to 100% of pipelines without additional efforts.

7 EVALUATION

7.1 Experimental setup

7.1.1 Query/Pipeline corpus. We evaluate MagicPush on both relational queries (TPC-H [17]) and real-world data science pipelines. Because TPC-H queries have already been well optimized by existing systems, the goal is to see whether MagicPush is able to achieve the same pushdown optimization. For real-world data pipelines, we evaluate on Jupyter Notebooks from GitHub that use Pandas library. We build a prototype parser that translates Python code with Pandas API into MagicPush's syntax described in Section 3. The parser includes three components: (1) a static

analyzer to decide whether a Pandas API or UDF can be handled by MagicPush (described later in Section 7.3) since a small fraction may not be handled like discussed in Section 6; (2) a parser that translates Python script into MagicPush's syntax; and (3) a code generator that produces optimized pipeline in Python. This parser currently parses 120 Pandas APIs with common non-UDF parameters and embedded UDF specified as lambda function [6].

To collect the data processed by these pipelines, we use the automatic notebook-replay technique introduced in AutoSuggest [67], We use the automatic notebook-replay technique introduced in AutoSuggest [67], and collected the trace of replay, i.e., the sequence of functions executed. We use the longest dataflow path in a notebook as the data processing pipeline, removing side paths like checking intermediate results. The final output of the pipeline is often used to build machine learning models or visualized to understand specific aspects of the data. We randomly sampled from replayable notebooks for coverage and performance evaluation: how many operators and UDFs can MagicPush cover and how much performance gain can it obtain by pushing predicates down.

MagicPush is implemented in Python with Z3 [20] SMT solver, and the data science pipelines run in-memory with Pandas versioned 1.1.4. We perform our evaluations on a server with a 2.4GHz processor and 64GB memory.

7.1.2 Baseline. We compare MagicPush with PostgreSQL query optimizer on relational queries. We also construct a rule-based baseline where rules are collected from existing systems or prior research [11, 23, 31, 32, 42, 44, 49]. These rules cover all common relational operators and some popular non-relational operators, listed as follow.

(1) If Op is a projection, then F can be pushed through Op; if Op renames columns while projection, F can be pushed down by replacing column names correspondingly; if Op sets new a column with a UDF, F can be pushed if it does not involve the new column, otherwise, replace the new column in F with the UDF only if the UDF is an invertible function on a single column. However, prior work does not mention how to check if a function is invertible, so we check invertibility using SMT solver[2] and simply treat it as not invertible if it contains black-box function [42, 44].
(2) If Op is a groupby, push F if all columns involved are group-by columns (same rule applies for distinct); if the aggregation is `max(C)` and F is $max_C \geq v$ (where `C` is a column and `v` is a constant), then set G to be $C \geq v$, and similarly for `min` aggregation [23, 42, 44].
(3) If Op is an intersection, union or sorting operator, then F can be directly pushed down [42, 44].
(4) If Op is an equijoin and all columns in F belong to one table, push F to that table; if F is in conjunctive normal form (CNF), then push the literals in CNF that involve only columns from one table; if the CNF involves literals involving columns from both tables, then keep F to filter the join output and treat the pushed literals as additional predicates [11, 32, 42, 49].
(5) If Op is an equijoin, F in disjunctive normal form and each literal involves both tables (e.g., predicate `(n1.name=='NATION1' & n2.name=='NATION2') | (n1.name=='NATION2' & n2.name=='NATION1')` in TPC-H Q7), add additional predicate to each table by rewriting F to remove comparisons on foreign columns (e.g., `(n1.name=='NATION1') | (n1.name=='NATION2')` to table `n1` and `(n2.name=='NATION1') | (n2.name=='NATION2')` to table `n2`), while still keeping the original predicate after Op [11, 32].
(6) If Op is a left outer join and F only involve columns from the left table, push F to the left table; if F only involves the join column, push to both tables. Same for right join [11, 42]. only involve columns

[2]Because the operator is required to take only one column, so the UDF takes in a single value for each row, the check is sound.

(7) If *Op* is a semi-join or anti-semi join (which corresponds to *RowIterPipeline* operator that returns a boolean value for each row to be used as filter) and all columns in F belong to the left table, push F to the left table [42].
(8) If *Op* has subquery (i.e.,*RowIterPipeline*), push down the predicate in subquery that involve only tables in the subquery (e.g., `r.name=='REGION'` in TPC-H Q2) but not tables from the outer query (e.g., `p.partkey==ps.partkey` in Q2 where `p.partkey` is from outer table `part`). Any predicate pushed to the table in the subquery will be disjuncted with other predicates on the same table obtained from the outer query [11, 23, 49].
(9) If *Op* is a pivot or unpivot and all columns in F are index columns, then F can be pushed down [31].

7.2 Evaluation on TPC-H

We observe that existing system like PostgreSQL already pushes predicates down in the query plan, therefore in this experiment, we check whether MagicPush is able to achieve the same optimization as PostgreSQL. We manually construct unoptimized query plans as follow: we ask PostgreSQL to generate a query plan, rewrite it into a pipeline, and then move all predicates in the query plan to their original position in the SQL query. Doing so often results in plans where joins operate on unfiltered tables, while other non-join predicates and aggregates are later executed on the joined result. We construct 18 unoptimized plans that are different from what PostgreSQL generated (except for Q1, Q6, Q14, and Q18 where predicates cannot be moved), and we ask MagicPush and the baseline described in Section 7.1.2 to work on these unoptimized plans.

Both MagicPush and the baseline are able to push down predicates properly for all 18 unoptimized plans, producing the same plan as what PostgreSQL originally generated. It is not surprising for the baseline as many rules are particularly designed for relational queries like TPC-H. However, MagicPush's approach without any particular rule tailored to relational queries is still able to **find and verify all the previously-discovered pushdown opportunities** in TPC-H, including non-trivial pushdowns like adding additional predicates to optimize Q7 and Q19 (covered by rule (5)).

7.3 Coverage of real-world operators and UDFs

We randomly sample 1000 operators and 1000 UDFs to analyze the coverage ratio regarding the limitation discussed in Section 6. For UDFs, it cannot be covered by MagicPush if it either has a different input-output type than specified in Table 1, or includes black-box functions not allowed other than scenarios listed in Table 2. For operators, it cannot be covered if it either belongs to unhandled operator types like transpose or embed uncovered UDFs.

MagicPush's parser includes a static analyzer that checks whether a UDF can be handled by MagicPush, which we leverage to examine MagicPush's UDF coverage. It analyzes the type of input and output to any UDF involved. Furthermore, it examines if any global variable is used in the UDF (e.g., the UDF in Figure 3 uses a global variable `df_old`) and treats these variables as UDF's implicit input. To check for black-box functions, it compiles the UDF into bytecode to see if it uses any global variable that are imported library other than `Pandas` library (e.g., `re.sub(...)` where `re` is of type "module"), or if it contains any function call outside the list of python built-in functions [14] (e.g., `x.strip()` or `unidecode(x)` where both `strip` and `unidecode` are not built-in functions). Any non built-in library functions are treated as black-box functions.

Table 3 shows the result. 99.7% UDFs can be handled by MagicPush, only 3 include black-box functions in unsupported operators like *GroupBy* and *RowIterPipeline*. This does not mean that library functions are rarely used: indeed 18.7% UDFs call library functions, only that most are embedded in operators allowing uninterpretable functions like *RowTransform*, *Filter* and *RowExpand*.

Table 3. Coverage on 1000 sampled operators and UDFs.

	covered	unhandled operator	other input-output type	unhandled black-box function
UDF	**99.7%**	-	0%	0.3%
Operator	**92.9%**	6.7%	0%	0.4%

Table 4. Profile of 200 sampled pipelines.

	# operators	input data size	scaled data size	original running time	scaled running time
min	5	0.01MB	10MB	0.005sec	0.5sec
max	100	390MB	4GB	183sec	964sec
average	17	22MB	0.9GB	2.6sec	35.2sec
median	13	2.8MB	1GB	0.5sec	6.9sec

For operator coverage, 92.9% operators are covered and the rest are mainly unhandled operators that predicates cannot be pushed through.

7.4 Evaluation on real-world data pipelines

To evaluate the actual performance gain, we sampled 200 pipelines that contain at least one `Filter` operator and can be compiled into MagicPush's syntax by our parser. Among them, 95 cannot be optimized because 1) the filters are invoked immediately after data loading (i.e., already "down" at loading) so there is no room for optimization, or 2) the predicate cannot be pushed down to data loading by by either MagicPush or the baseline. We report the performance for the remaining 105 pipelines that can be optimized.

Table 4 shows the profile of the sampled pipelines. The size of data processed by these pipelines ranges from 0.01MB to 390MB and the number of operators ranges from 5 to 100. Because many pipelines only process a small amount of data, the running time is short (a median of only 0.5sec) with a large variance. We further run them on scaled data where we duplicate each original input until reach 1GB.[3] If the pipeline fails to run due to out-of-memory, we scale it down to 100MB or 10MB. Running on the scaled data takes longer (a median of 28sec) and the running time is more stable.

Figure 9(a) shows the amount of data reduced by the pushed predicate, measured by row count. The result shows that pushdown opportunities are abundant in real-world pipelines: MagicPush is able to push down predicate to the pipeline input for 105 pipelines while the baseline can optimize for a subset of 74. In 42 pipelines MagicPush reduces more data than the baselines by 46% on average, up to 99% (variance=0.1). We compare the predicates after pushed down and find that **in all cases MagicPush's predicate is either strictly more selective (i.e., conjuncting more comparisons) than the baseline (in 42 pipelines) or the same (in the remaining 63 pipelines), but never worse**. Furthermore a statistical significance test on the data (where the null hypothesis states that MagicPush is no better than baselines) produces a p-value of 1.4e-8, allowing us to reject the null hypothesis and showing a strong statistical significance of our results. This result shows that MagicPush is able to discover all pushdown opportunities found by prior rules and often more opportunities where the baseline would miss due to the limited patterns.

[3]Duplicating data ensures that UDFs can run successfully – alternatives such as mixing values sometimes makes the pipeline fail due to implicit dependency assumed on the data, e.g., "df.apply(lambda row: row['c1']+row['c2'] if notnull(row['c1']) else NULL)" assumes the value of column c2 is not NULL if c1 not NULL; mixing values randomly will likely crash the UDF.

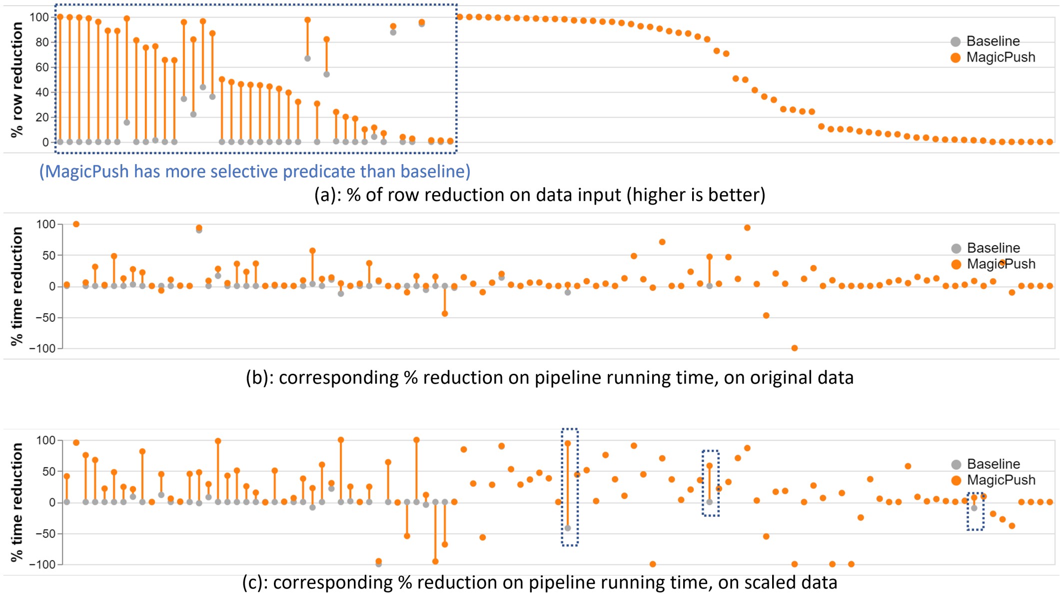


Fig. 9. Improvement on real-world data pipelines; each point in x-axis represents a pipeline, sorted by the gap of row reduction between MagicPush and the baseline in (a).

Figure 9(b)(c) shows the end-to-end performance gain on both the original and scaled data. On the 42 pipelines where MagicPush can further reduce data loading than baseline, it achieves better performance gain on 35 pipelines on the original data and 38 pipelines on the scaled data, further reducing pipeline running time by 10% on average (up to 53%, with a variance of 0.02) on the original data, and 26% on average (up to 99%, with a variance of 0.04) on the scaled data. The statistical significance tests on the two running time results are $p = 0.0013$ and $p = 0.0001$, respectively, again confirming the significance of our results.

Because we still load the entire data and use Pandas API to run the pushed filter, the filter itself often becomes slower after pushdown as it processes more data using the same API. The performance gain is only significant when the predicate is very selective such that other operators can be greatly accelerated. We expect the benefit to be much larger on systems that support processing predicates during data loading like Mison [45].

In a few cases pushing down predicate actually makes the pipeline slower when the predicate is not selective, or includes UDF that takes a long time to finish. Whether to push the predicate down or pull it up should depend on the cost estimation [44], which can be very challenging, particularly under the wide existence of library functions. We leave it as future work.

7.5 Case studies

In addition to the non-trivial pushdowns listed in Section 2, we present more interesting cases observed from real-world pipelines.

Case 1. Pushing predicates through outer joins is usually not straightforward. Listing 3 shows an example where the original code includes a left outer join (line 5) followed by a predicate F involving columns from both tables (line 7). This conjuntive predicate can be split and added to each table, which reduces about half of the data loaded. Different from inner join where such predicate F can be pushed down directly (i.e., F removed after pushdown), with left outer join the F remains as the pushdown actually returns a superset. Whether a predicate can be pushed becomes even more challenging when null-value is involved. For instance, the predicate shown on Line 8 cannot be pushed down at all, since both joinable and non-joinable rows are both used in this predicate.

Listing 3. Pushdown with left join

```
1  # corresponding G:
2  df_2017 = df_2017['act_2017']>0.5
3  df_2018 = df_2018['act_2018']>0.5
4  # left outer join operator
5  df = df_2017.merge(df_2017, on='state', how='outer')
6  # predicate F to be pushed down
7  df = df[(df['act_2017']>0.5) & (df['act_2018']>0.5)]
8  #df = df[(df.apply(lambda r: r['v_2017'] if pd.isnull(r['v_2018']) else r['v_2018'])>1)]
```

Case 2. MagicPush handles many non-relational operators that has not been studied before, like the popular `get_dummies` operator that converts row into one-hot encoding (a common step in ML), with an example shown in Listing 4.

Listing 4. Pushdown with getdummies

```
1  # corresponding G:
2  df = df[df['status']=='Left']
3  # get_dummies operator
4  dummy = df.get_dummies(df['status'])
5  # predicate F to be pushed down
6  count = dummy[dummy['Left']==1].count()
```

Case 3. Occasionally MagicPush returns predicate that has the same selectivity as the baseline but more efficient when evaluated. Listing 5 shows one example which includes a row-transform operator (line 5) followed by a predicate F (line 7) which involve rows being transformed earlier. MagicPush is able to push F down unchanged as the row-transformation does not affect the evaluation of F and F passes the verification. In contrast, the baseline always returns a predicate replacing the column in F with the transformation using rule (a) (line 3), which is much slower. Three pipelines, as circled out in Figure 9(c), benefit from such more-efficient predicate, improving the performance by up to 25× compared to the baseline.

Listing 5. Pushdown returning simpler predicate

```
1  # corresponding G:
2  df = df[df['Country'] != 'U.S.']
3  # df=df[df['Country'].replace('HongKong','China')!='U.S']
4  # row-transform operator
5  df['Country'] = df['Country'].replace('HongKong','China')
6  # predicate F to be pushed down
7  df = df[df['Country'] != 'U.S.']
```

7.6 MagicPush's running time

We present MagicPush's end-to-end running time to optimize a query or a pipeline (i.e., from taking a Python script to producing the optimized pipeline in Python) in Figure 10. It is very efficient, taking <0.2sec for TPC-H query (0.1sec on average) and <0.8sec for 98 pipelines (0.33sec on average).

There are only three "outlier" pipelines that MagicPush takes a while to optimize, due to the complicated UDF and large symbolic table in verification. The slowest is due a UDF that contains 14 branch conditions, shown in Listing 6. The correct predicate G is a disjunction of 2 branch condition picked from 14, and MagicPush exhaustively tried over 1000 candidates before finding the correct one. The other two slow pipelines are due to the *Pivot* operators. Because for *Pivot* the base-case table size is the number of pivoted columns in the output table, when the output table is wide (over 30 columns in these pipelines), MagicPush will verify on a large symbolic tables containing over 30 symbolic rows, leading to a slow verification. This can be accelerated by exploiting symmetry

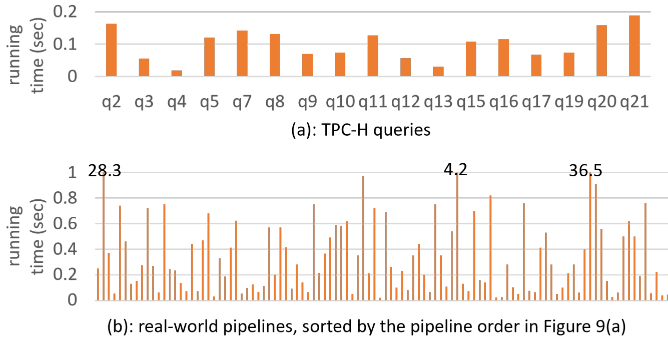


Fig. 10. MagicPush's running time

among symbolic rows to reduce the base-case table size proposed by Wang et. al. [62], and we leave this optimization as future work.

Listing 6. An UDF including many branch conditions

```
# corresponding G:
df = df[df.apply(lambda row: 'Software' in row['CategoryGroup']
or 'Data' in row['CategoryGroup'])]
# UDF
df['Category'] = df.apply(lambda row:
  'Finance' if 'Finance' in row['CategoryGroup'] else
  'Software' if 'Software' in row['CategoryGroup'] else
  ... # other 12 similar branches
  else 'Unknown')
# predicate F to be pushed down
df = df[(df['Category']=='Software') | (df['Category']=='Data')]
```

8 RELATED WORK

Predicate pushdown. Rules to pushdown predicate are used in nearly all database systems: Postgres [40], Spark [24], Synapse [48], SQL Server [34], Hadoop [35], Vertica [57], AnalyticDB [70], to name a few. Various new systems have been developed to leverage the power of predicate: Crystal [33] is a caching system incorporating query optimization like predicate pushdown, Qd-tree [69] leverages pushed predicate to better organize data on disk, FlexPushdownDB [68] explores pushdown on storage-disaggregation architecture, diP [42] shows how to use indexes to push additional predicate, Mison [45] implements an efficient JSON parser for data loading that leverages predicate. While predicate pushdown has been widely studied and supported in DBMS, MagicPush extends its power to also optimize for ETL pipelines. Other work like SIA [74] shares similar optimization by synthesizing additional predicate on top of existing ones, but focuses only on predicate equivalence instead of pushdown equivalence over UDFs and operators.

Dataframe algebra, optimizations, and systems. Modin [52, 53] proposes dataframe algebra that captures commonality among various Pandas operators such that implementing a few core operators can cover a wide range of APIs. This algebra is also helpful in deciding a decomposed execution of an operator to run it in parallel. We also propose core operators summarized based on logical functionalities for predicate pushdown, compared to Modin's summarized from shared implementation.

Dataframe systems are becoming increasingly prevalent due to the popularity of Pandas in EDA: Koalas [4] runs Pandas on Spark, Dask [2] scales in-memory Pandas with multi-threads, Modin [53] parallelizes operator execution with Ray, Ibis [3] translates Pandas APIs into a few

database backends, MagPie [41] decides which backend to choose in a cloud environment. These systems scale out the execution of each single operator or running it on a relational database. As MagicPush optimizes operator execution order, it is orthogonal to these systems and can be combined together (e.g., applying MagicPush first to reorder operators before running on optimized dataframe systems) to improve pipeline performance.

Deciding query equivalence. SQL query equivalence is a problem that has been extensively studied in the database theory community [25, 29, 30]. Recently Chu et al. [28] proposes formalizing queries with U-semiring and built Cosette with Coq to search for equivalence proof. Many other systems apply or improve Cosette: WeTune [65] discovers rewrite rules from a workload and verifies these rules automatically, SPEC [73, 75] extends to verify queries with different structures by trying to "normalize" the query structure, and also extends U-semiring formalization to handle NULL values. These techniques show great potential for various query optimization but are restricted to relational operators and cannot be extended to UDFs. Furthermore, each query pair to be checked often requires a distinct proof, making full automatic verification challenging. In contrast, MagicPush focuses only on predicate pushdown, fully leveraging the structural similarity between queries (e.g., the operator *Op* is the same) to construct a single proof for each operator category which is often too complicated to be automatically searchable. However, it greatly extends the scope of operators and UDFs (like UDF that includes a subpipeline) and achieves great coverage on real-world pipelines.

Other than read-only SQL, Mediator [63] reasons about database applications with updates, Wang et. al [64] proposes FGH-rules to optimize recursive Datalog programs, and SparkLite verifier [37] checks MapReduce program equivalence. The invariants proposed in FGH-rules and SparkLite share the same spirit with our pre-condition for the small-model property. However, FGH-rules work on single-recursion programs with different loop bodies and SparkLite's invariant works with different aggregation functions; both cannot be directly used to solve predicate pushdown.

Symbolic execution in DMBS. Researchers have leveraged symbolic execution to optimize queries and test database functionalities: Chestnut [66] and Cozy [46] generate new data layout and verify query plan on the layout, Blitz [43, 56] synthesizes UDO from spark programs to better parallel query execution, and many efforts to generate test cases for database applications [50, 51, 59, 61]. These work either admit bounded verification as a limitation or only returns example with no need for correctness proof.

Other query optimization with SQL UDFs. Prior work explored optimizing queries with UDF 1) by converting them into SQL queries and integrating them into other SQL components, like Froid [55], QBS [26] and CLIS [71]; 2) by analyzing properties of UDF like data partition to avoid unnecessary data shuffling in parallel query execution [21, 38, 72]; 3) by optimizing the UDF compilation to generate more efficient execution in DBMS runtime, like Tuplex [58], and YeSQL [36]; etc. We discussed the scope of UDFs MagicPush focuses on and compared it with SQL UDFs in Section 3. The difference is slight and MagicPush can be easily extended to cover a wide range of SQL UDFs, and we believe its potential in many other optimizations with UDFs (like parallel query execution) beyond predicate pushdown.

9 CONCLUSION

In this work, we propose MagicPush to decide predicate pushdown for data pipelines involving relational and non-relational operators with embedded UDFs. We show that MagicPush's novel search-verification method outperforms traditional pushdown rules and discovers more pushdown opportunities on TPC-H queries and sampled real-world pipelines, while providing a full correctness guarantee of its optimization. We believe MagicPush's approach will guide many other systematic query optimizations on UDFs for future research.

REFERENCES

[1] [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/.
[2] [n. d.]. Dask: a flexible library for parallel computing in Python. https://docs.dask.org/en/stable/.
[3] [n. d.]. Ibis: expressive analytics in Python at any scale. https://ibis-project.org/docs/3.2.0/.
[4] [n. d.]. Koalas: pandas api on apache spark. https://koalas.readthedocs.io/en/latest/.
[5] [n. d.]. Pandas apply API. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html.
[6] [n. d.]. Pandas apply API with lambda function as parameter. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html.
[7] [n. d.]. Pandas: data analysis and manipulation library in Python. https://pandas.pydata.org/.
[8] [n. d.]. Pandas explode API, convert one row to multiple rows. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.explode.html.
[9] [n. d.]. Pandas get_dummies API, convert a column to one-hot encoding. https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html.
[10] [n. d.]. Pivot and Unpivot operator. https://learn.microsoft.com/en-us/sql/t-sql/queries/from-using-pivot-and-unpivot?view=sql-server-ver16.
[11] [n. d.]. PostgreSQL query optimizer. https://www.postgresql.org/docs/current/planner-optimizer.html.
[12] [n. d.]. Power Query data flow. https://learn.microsoft.com/en-us/connectors/dataflows/.
[13] [n. d.]. Proof for pushdown correctness and SMP of pre-condition in Coq. https://github.com/predicate-udf/pushdown-smp-proof.
[14] [n. d.]. Python built-in functions. https://docs.python.org/3/library/functions.html.
[15] [n. d.]. Python core statements. https://docs.python.org/3/reference/grammar.html.
[16] [n. d.]. Tableu: build and organize your flow. https://help.tableau.com/current/prep/en-us/prep_build_flow.htm.
[17] [n. d.]. TPC-H benchmark. https://www.tpc.org/tpch/.
[18] [n. d.]. Trifacta data pipeline. https://www.trifacta.com/blog/orchestrate-data-pipelines-on-trifacta-using-plans/.
[19] [n. d.]. Uninterpreted functions in Z3. https://microsoft.github.io/z3guide/docs/logic/Uninterpreted-functions-and-constants/.
[20] [n. d.]. The Z3 theorem prover. https://github.com/Z3Prover/z3.
[21] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-Optimizing Data-Parallel Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, USA, 21. https://dl.acm.org/doi/10.5555/2228298.2228327
[22] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 1205–1220. https://doi.org/10.1145/3183713.3196891
[23] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-Based Query Transformation in Oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. 1026–1036. https://dl.acm.org/doi/10.5555/1182635.1164215
[24] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data (SIGMOD)*. 1383–1394. https://doi.org/10.1145/2723372.2742797
[25] Surajit Chaudhuri and Moshe Y. Vardi. 1993. Optimization of Real Conjunctive Queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 59–70. https://doi.org/10.1145/153850.153856
[26] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. *PLDI* 48, 6 (2013), 3–14. https://doi.org/10.1145/2491956.2462180
[27] Shumo Chu, Chenglong, Konstantin Weitz, and Alvin Cheung. 2017. Demonstration of the Cosette Automated SQL Prover. In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*.
[28] 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 (PLDI)*. 510–524. https://dl.acm.org/doi/10.1145/3140587.3062348
[29] Sara Cohen. 2009. Equivalence of queries that are sensitive to multiplicities. *The VLDB Journal* 18, 3 (2009), 765–785. https://link.springer.com/article/10.1007/s00778-008-0122-1
[30] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2007. Deciding Equivalences among Conjunctive Aggregate Queries. *J. ACM* 54, 2 (apr 2007), 5–es. https://doi.org/10.1145/1219092.1219093
[31] Conor Cunningham, César Galindo-Legaria, and Goetz Graefe. 2004. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. *Proc. VLDB Endow.* 30 (2004), 998–1009. https://dl.acm.org/doi/abs/10.5555/1316689.1316775
[32] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[33] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2432–2444. https://doi.org/10.14778/3476249.3476292
[34] Kevin Feasel. 2020. Using Predicate Pushdown to Enhance Query Performance. In *PolyBase Revealed*. Springer, 95–125. https://link.springer.com/chapter/10.1007/978-1-4842-5461-5_4
[35] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1295–1306. https://doi.org/10.14778/2732977.2733002
[36] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. 2022. YeSQL: "You Extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2270–2283. https://doi.org/10.14778/3547305.3547328
[37] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. Verifying equivalence of spark programs. In *International Conference on Computer Aided Verification (CAV)*. Springer, 282–300. https://link.springer.com/chapter/10.1007/978-3-319-63390-9_15
[38] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 121–133. https://dl.acm.org/doi/10.5555/2387880.2387893
[39] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops Using Custom Aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 559–573. https://doi.org/10.1145/3318464.3389736
[40] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. *SIGMOD Rec.* 22, 2 (jun 1993), 267–276.
[41] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas C. Müller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *Conference on Innovative Data Systems Research (CIDR 2021)*.
[42] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (nov 2019), 252–265. https://doi.org/10.14778/3368289.3368292
[43] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating Super-Operators in Big-Data Query Optimizers. *Proc. VLDB Endow.* 13, 3 (nov 2019), 348–361. https://doi.org/10.14778/3368289.3368299
[44] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query Optimization by Predicate Move-Around. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. 96–107. https://dl.acm.org/doi/10.5555/645920.672839
[45] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (jun 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416
[46] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE) (ICSE '18)*. 958–968. https://doi.org/10.1145/3180155.3180211
[47] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast Synthesis of Fast Collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (PLDI '16)*. 355–368. https://doi.org/10.1145/2980983.2908122
[48] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New Query Optimization Techniques in the Spark Engine of Azure Synapse. *Proc. VLDB Endow.* 15, 4 (dec 2021), 936–948. https://doi.org/10.14778/3503585.3503601
[49] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of magic-sets in a relational database system. *ACM SIGMOD Record* 23, 2 (1994), 103–114. https://doi.org/10.1145/191839.191860
[50] Kai Pan, Xintao Wu, and Tao Xie. 2013. Automatic Test Generation for Mutation Testing on Database Applications. In *ASE*. 111–117. https://dl.acm.org/doi/10.5555/2662413.2662439
[51] Kai Pan, Xintao Wu, and Tao Xie. 2014. Guided Test Generation for Database Applications via Synthesized Database Interactions. *ACM Trans. Softw. Eng. Methodol.* 23, 2, Article 12 (apr 2014), 27 pages. https://doi.org/10.1145/2491529
[52] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807
[53] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proc. VLDB Endow.* 15, 3 (nov 2021), 739–751. https://doi.org/10.14778/3494124.3494152
[54] Amir Pnueli, Yoav Rodeh, Ofer Strichman, and Michael Siegel. 2002. The small model property: How small can it be? *Information and computation* 178, 1 (2002), 279–293. https://dl.acm.org/doi/abs/10.1016/S0890-5401%2802%2993175-5

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

[56] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP) (SOSP '17)*. 631–646. https://doi.org/10.1145/3132747.3132773

[57] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1196–1207.

[58] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 1718–1731. https://doi.org/10.1145/3448016.3457244

[59] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. 2015. TesMa and CATG: automated test generation tools for models of enterprise applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 717–720. https://dl.acm.org/doi/10.5555/2819009.2819147

[60] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic query exploration. In *International Conference on Formal Engineering Methods*. Springer, 49–68. https://doi.org/10.1007/978-3-642-10373-5_3

[61] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446. https://dl.acm.org/doi/10.5555/1939141.1939165

[62] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2018. Speeding up Symbolic Reasoning for Relational Queries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 157 (oct 2018), 25 pages. https://doi.org/10.1145/3276527

[63] 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. https://doi.org/10.1145/3158144

[64] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Progam Synthesis. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 79–93. https://doi.org/10.1145/3514221.3517827

[65] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)* (Philadelphia, PA, USA) *(SIGMOD '22)*. 94–107. https://doi.org/10.1145/3514221.3526125

[66] Cong Yan and Alvin Cheung. 2019. Generating Application-Specific Data Layouts for in-Memory Databases. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1513–1525. https://doi.org/10.14778/3342263.3342630

[67] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1539–1554. https://doi.org/10.1145/3318464.3389738

[68] Yifei Yang, Matt Youill, Matthew 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 (jul 2021), 2101–2113. https://doi.org/10.14778/3476249.3476265

[69] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 193–208. https://doi.org/10.1145/3318464.3389770

[70] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-Time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2059–2070. https://doi.org/10.14778/3352063.3352124

[71] 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. https://doi.org/10.1145/3485489

[72] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 22. https://dl.acm.org/doi/10.5555/2228298.2228328

[73] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Wu Jinpeng. 2020. A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. (2020). https://doi.org/10.48550/arXiv.2004.00481 arXiv:arXiv:2004.00481

[74] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2021. SIA: Optimizing Queries Using Learned Predicates. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2169–2181.

https://doi.org/10.1145/3448016.3457262

[75] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability modulo Theories. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1276–1288. https://doi.org/10.14778/3342263.3342267

Received October 2022; revised January 2023; accepted February 2023