Optimizing Big-Data Queries Using Program Synthesis

Matthias Schlaipfer
TU Wien
Vienna, Austria
mschlaipfer@forsyte.at

Kaushik Rajan
Microsoft Research
Bangalore, India
krajan@microsoft.com

Akash Lal
Microsoft Research
Bangalore, India
akashl@microsoft.com

Malavika Samak
MIT CSAIL
Cambridge, USA
malavika@csail.mit.edu

ABSTRACT

Classical query optimization relies on a predefined set of rewrite rules to re-order and substitute SQL operators at a logical level. This paper proposes Blitz, a system that can synthesize efficient query-specific operators using automated program reasoning. Blitz uses static analysis to identify sub-queries as potential targets for optimization. For each sub-query, it constructs a template that defines a large space of possible operator implementations, all restricted to have linear time and space complexity. Blitz then employs program synthesis to instantiate the template and obtain a data-parallel operator implementation that is functionally equivalent to the original sub-query up to a bound on the input size.

Program synthesis is an undecidable problem in general and often difficult to scale, even for bounded inputs. Blitz therefore uses a series of analyses to judiciously use program synthesis and incrementally construct complex operators.

We integrated Blitz with existing big-data query languages by embedding the synthesized operators back into the query as User Defined Operators. We evaluated Blitz on several production queries from Microsoft running on two state-of-the-art query engines: SparkSQL as well as Scope, the big-data engine of Microsoft. Blitz produces correct optimizations despite the synthesis being bounded. The resulting queries have much more succinct query plans and demonstrate significant performance improvements on both big-data systems (1.3x — 4.7x).

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132773

CCS CONCEPTS

• **Information systems** → **Query optimization**; *Parallel and distributed DBMSs*; *Query operators*; • **Theory of computation** → *Program analysis*; Program specifications;

KEYWORDS

Program Synthesis, Query Optimization, User-Defined Operators

ACM Reference Format:
Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of SOSP '17*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3132747.3132773

1 INTRODUCTION

Big-data analytics is typically performed by writing and executing queries in SQL-like languages [1, 20, 28] supported by systems such as Hadoop [23], Scope [2] and Spark [25]. The queries are compiled to an execution plan that consists of a DAG of map-reduce-like stages. The query compilation happens in three phases. First, operators are transformed and reordered at the logical level, for example, SQL operators are substituted with other equivalent SQL operators and filtering operators are pushed up closer to the source of the data. Next, efficient physical implementations are chosen for each operator; typical SQL operators lend to highly-optimized low-complexity implementations. Finally, operators are grouped together into stages as long as the composition continues to have low complexity. The generated plan is then executed on a distributed system such that each stage runs in parallel. The data between stages is shuffled over the network.

We analyzed the generated plan for several queries from different execution engines and noticed the following trend in a significant fraction of them.

(1) Many of the plans, represented as a DAG of stages, had a single stage that dominated significant parts of the query.[1] In other words, the number of stages whose input is functionally dependent on the output of a single stage is high.
(2) Furthermore, in the final data that is produced by the dominated sub-query, each row depends on only a subset of the rows of the output of the dominating stage.

These observations imply a possible query transformation to replace the entire dominated sub-query using a mapper that identifies the right group of rows and a reducer that applies a function to each group to produce the correct output. Such an optimization can have significant benefits because it reduces the number of stages, thus decreasing the amount of data that needs to be shuffled between machines. However, the reducer function must be efficient in order to process each mapped group on a single machine. For instance, naively executing the sub-query itself inside the reducer function would almost always blow up or run out of memory.

This paper introduces Blitz, a system that employs automated program reasoning to synthesize low complexity operators. Blitz extends an existing, but rarely used, analysis called groupwise analysis [3] to a formal static analysis that can identify sub-queries with the properties mentioned above. It then uses program synthesis to construct an efficient operator with linear time and space complexity, at the most relying on sorting features that are provided by most existing query-processing engines.

Program synthesis is the problem of constructing executable code given its input-output specification. In our setting, the specification comes from the semantics of the original query and the linear-time operator (modulo sorting) is the desired output. Program synthesis is an undecidable problem in general, and computationally very demanding in practice. Often, it is limited to very simple or very small synthesis tasks. Unsurprisingly, using synthesis naively was insufficient to optimize many of the queries in our benchmark set. To make synthesis practical, Blitz uses several auxiliary analyses that impose restrictions on the structure of the synthesized code to reduce the search space of possible implementations, while still allowing feasible solutions for most queries that we would like to optimize.

Blitz fixes the high-level structure of the operator by bounding the kinds of loops, branching and control flow that it can have. It further bounds the amount of local state that the operator can keep. It analyzes the input query to extract expressions, predicates and aggregation functions that will likely be reused in the operator. This information is used to generate a partial program, i.e., a program with *holes*. Next, Blitz employs an off-the-shelf program synthesis tool to fill these holes with code such that the resulting operator matches the semantics of the input query.

One limitation of Blitz is that it only guarantees partial soundness. Current synthesis tools are able to guarantee that the synthesized implementation meets its specification only up to a fixed bound on the size of the input. We therefore translate the synthesized operator back to the source language as a query with User-Defined Operators (UDOs) and validate manually that the optimization is indeed correct. Our experience with such validation has been surprisingly positive and we discuss this in more detail in §8.

We evaluated Blitz on several production queries from Microsoft. We analyzed all jobs from a day's run on one of Microsoft's clusters and found that about one-third of them have stages that dominate three or more other stages. We picked a few such queries that were long-running, optimized them using Blitz, and compared their performance against the original queries on Spark and Scope. Blitz significantly reduces the execution time of the queries on both big-data systems. Queries optimized by Blitz attain an average speedup of 2×, have 65% fewer stages, require less than half the cumulative CPU time and shuffle up to 75% less data.

The rest of this paper is organized as follows. Section 2 gives an overview of Blitz on two example queries. Section 3 sets up the formal notation and explains groupwise analysis. Section 4 describes the UDO template that we use for synthesis. Section 5 describes the architecture of Blitz. Section 6 presents additional analyses used by Blitz to help the synthesis task scale. Section 7 presents our experimental evaluation. Section 8 discusses practical challenges with using our system. Section 9 discusses related work.

2 MOTIVATING EXAMPLES AND OVERVIEW

We motivate our approach with two example queries. The first is from TPCx-BigBench [21], a standard benchmark suite for evaluating big-data systems. We encoded a simplified version of the query BigBench8, shown in Figure 1.[2] The query operates over a web-click stream with 4 fields:

```
wcs(ts:int, user:int, type:string, sales:int).
```

The column `ts` is the (unique) timestamp of the web click. Column `user` identifies the user, column `type` represents the transaction type ("buy" or "review"), and column `sales` is

[1] In the query plan DAG, a node (or stage) N_1 dominates node N_2 if all paths from the source node of the DAG to N_2 pass through N_1.

[2] The complete query operates on a join of two tables to construct `wcs` and has an additional predicate in `V1`. However, it has similar performance characteristics and we obtain similar speedups with Blitz.

```
VIEW V1 =
  SELECT s1.user,
         s1.sales,
         s1.ts AS bts,
         s2.ts AS rts
  FROM wcs AS s1 JOIN wcs AS s2
  ON s1.user = s2.user
  WHERE s1.type = "buy"
  AND   s2.type = "review"
  AND   s1.ts > s2.ts;

VIEW V2 =
  SELECT user, rts, MIN(bts) AS mts
  FROM V1
  GROUP BY rts, user;

VIEW V3 =
  SELECT ar.user, ar.sales
  FROM wcs AS ar SEMI JOIN V2 AS bia
  ON  ar.bts = bia.mts
  AND ar.user = bia.user;
```

Figure 1: A simplified version of the query BigBench8

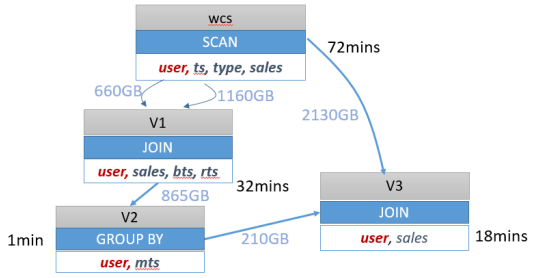

Figure 2: Execution plan for BigBench8

the sales amount associated with the click. The query aims to find buys where the customer checked online reviews just before making the purchase: table `V1` selects all buys after reviews using a self-join on the `wcs` table, table `V2` finds the smallest purchase timestamp and `V3` discards all rows except ones with this least timestamp.

Figure 2 shows the execution plan for this query generated by Scope, along with some runtime statistics. Note that all the stages in the plan are dominated by the top-most stage. The query plan is quite inefficient. It performs two expensive joins and shuffles data redundantly between stages. As a result, a large amount of time is spent in the dominated stages. We now apply our techniques to optimize this query.

We first notice that the final output for a particular user can only be influenced by the rows of the same user in input `wcs` table. This suggests partitioning the input on the `user` column, processing each partition independently and

```
TRANSFORM wcs
PARTITIONED BY user
SORTED BY ts
USING udo

proc udo(user, List rows)
  // assumes sorted([ts], rows)
  flag ← False
  foreach row ∈ rows
    if flag ∧ row.type == "buy"
      output(row)
      flag ← False
    if row.type == "review"
      flag ← True
```

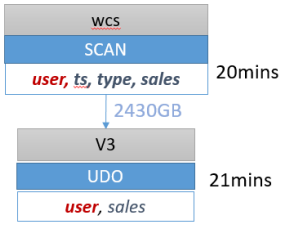

Figure 3: Optimized version of BigBench8 query

then putting the results back together. Such kinds of queries, which can be partitioned on a column and processed independently are called *groupwise* queries [3]. We use the groupwise analysis to find maximal sub-queries that serve as candidates for optimization. In this example, the entire query is considered for optimization.

One attempt at optimization, as suggested in the groupwise paper [3], could be to partition the input on `user` and then execute the query on each partition independently. This method will replace multiple simple stages with a single stage but that stage will have higher time and space complexity than any of the original stages. We tried this method and it either runs out of memory (when executed on Spark) or is much slower than the original query (8x slower when executed on Scope). Blitz instead uses program synthesis to produce the equivalent query shown in Figure 3. Blitz automatically figures out to partition on the `user` column, sort each partition on `ts` (timestamp) and execute the UDO shown in the figure to obtain the same output as the original query. Notice that the UDO has linear time complexity and requires just a constant amount of additional storage (the `flag` variable). In fact, the UDO can even be executed in a streaming fashion because it requires a single pass over the input, offering further benefits. The optimized query is 3× faster than the original query. In addition, it also uses much fewer resources; it shuffles much less data and completes execution with much fewer tasks. Finally, the synthesized query turns out to be simpler to understand than the original query.

It is clear that such an optimized query cannot be obtained from standard query optimizations that operate in a rule-based fashion, justifying the need for general-purpose program synthesis techniques.

We next motivate several of the challenges in scaling our technique to real-world production queries. Consider the query in Figure 4 that we obtained from Microsoft. We

have anonymized the column and table names for proprietary reasons but the structure of the query remains the same. The query is groupwise on columns a and b. However, a direct application of synthesis on the entire query does not scale. The synthesizer is unable to produce an answer even after a few days of execution. Blitz employs a series of query analyses to make synthesis more tractable.

First, it identifies that the query contains a **union** operator that simply puts together the results from each of its inputs. Blitz attempts to synthesize separate UDOs that produce tables Out1 to Out5 and then put the operators together in a rule-based fashion. For instance, it extracts a smaller query that computes Out1 and synthesizes an operator for it, as shown in the figure. Then it repeats this process for each of Out2 to Out5. Blitz attempts this strategy for N-ary operators whenever it knows that the N individual UDOs can be put together safely while preserving semantics and maintaining low UDO complexity (§6.1).

Second, Blitz further simplifies the sub-queries that compute Out1 to Out5. It eliminates the input and intermediate columns that are unnecessary for this sub-query (through a *taint analysis*, §6.2), and eliminates *redundant* columns (§6.3). These simplifications appear as crossed-out text in the figure. For instance, each of the columns m2 to m6 are irrelevant for Out1. Further, columns a and b always appear together, hence they can be condensed into the same column (thus, Blitz eliminates column b).

Finally, Blitz constructs the template for UDOs. A template is a *partial program*, i.e., a program with *holes* and the job of the synthesizer is to fill the holes. Fewer holes imply better scalability but a smaller set of possible implementations. Blitz constructs the template to restrict attention to linear-time operators that can only do a fixed number of passes over the input. It considers pre-sorting the input. Often we notice that sorting can drastically reduce the complexity of the UDO. If the synthesizer finds that sorting is useful, Blitz pushes sorting into the previous stage before the UDO is called. Further, Blitz restricts the form of predicates and expressions in the template: they can either contain predicates or expressions that appear in the input query, or do a simple manipulation of a finite number of flags and memoization variables.

The synthesized UDO is shown in the figure as the procedure udo2. Notice how udo2 uses sortedness on column c and the cnt1 aggregate for simulating "**COUNT**(**DISTINCT** c)". The synthesis takes around 5 minutes to finish. The optimized query runs 4× faster than the original query.

```
VIEW V =
 SELECT a, b, c,
  SUM(m1) AS m1,SUM(m2) AS m2,
  SUM(m3) AS m3,SUM(m4) AS m4,
  SUM(m5) AS m5,SUM(m6) AS m6
  FROM Input
  GROUP BY a, b, c;
VIEW Out1 =
 SELECT a, b
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE m1 > 100
 GROUP BY a, b;
VIEW Out2 =
 SELECT a, b
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE m2 > 100
 GROUP BY a, b;
VIEW Out3 =
 SELECT a, b
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE m3 > 100
 GROUP BY a, b;
VIEW Out4 =
 SELECT a, b
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE  m4 > 100
 GROUP BY a, b;
VIEW Out5 =
 SELECT a, b
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE  m5 > 100
 GROUP BY a, b;
VIEW Final =
 Out1 UNION Out2 UNION
 Out3 UNION Out4 UNION Out5;
```

```
VIEW V =
 SELECT a, ~~b~~, c,
  SUM(m1) AS m1,~~SUM(m2) AS m2~~
  ~~SUM(m3) AS m3~~, ~~SUM(m4) AS m4~~
  ~~SUM(m5) AS m5~~, ~~SUM(m6) AS m6~~
  FROM Input
  GROUP BY a, ~~b~~, c;
VIEW Out1 =
 SELECT a, ~~b~~
  COUNT(DISTINCT c) AS c
 FROM V
 WHERE m1 > 100
 GROUP BY a, ~~b~~;

TRANSFORM Input
PARTITIONED BY a
SORTED BY c
USING udo2
proc udo2(a, List rows)
  // assumes sorted([c], rows)
  first ← True
  foreach (c,m1) ∈ rows
    if first
      sum1 ← 0; cnt1 ← 0;
      oldC ← c; first ← False;
    if oldC == c
      sum1 ← sum1 + m1
    else
      if (sum1 > 100) cnt1++;
      sum1 ← m1; oldC ← c;
  if (sum1 > 100) cnt1++;
  if cnt1 > 0
    output(a, cnt1)
```

Figure 4: (Left) The complete query (Q2); (Right, top) Simplified sub-query that Blitz feeds to the synthesis tool; (Right, bottom) the generated UDO.

3 BACKGROUND

This section defines the SQL-like input language of Blitz. The language is kept simple to ease the presentation of our algorithms but it is powerful enough to encode all queries that we considered in our evaluation.

We represent a table as a list of records, where a record is of the form $h = \{a_1 : v_1, \ldots, a_n : v_n\}$. We write $h.a_i$ to access the value v_i in a record h at column a_i. Values are either integers or rationals, i.e., pairs of integers. We say that two records are equivalent if they contain the same columns and all corresponding values are equal. We use $\text{cols}(h) = \{a_1, \ldots, a_n\}$ to access the columns of a record or a list of records.

We support operators akin to their SQL equivalents that operate on tables, namely: selection, projection, renaming of columns, join, union and group-by. In contrast to SQL, however, each operator in our language imposes an order on the output records relative to the order in the input.

The semantics of the operators is standard. Selection (σ_φ) takes a predicate φ and uses it to filter the input. Projection (π_A) limits the output to the columns in A; it does not remove duplicates. Union (:::) appends one list to another without removing duplicates. Inner join ($\bowtie_\varphi$) takes two lists and

iterates over all pairs of rows. It outputs the concatenated pair subject to the filter predicate φ. Rename ($\rho_{A \mapsto A'}$) renames the columns in A to A' and leaves the remaining ones unchanged. The rename operator is necessary to avoid clashes between column names of different tables. Group-by (γ_{A,F_B}) takes a set of grouping columns A, and a set of aggregation functions over the aggregation columns B. It partitions the input on all distinct values that appear in columns A, and for each partition it computes the aggregation and concatenates it with the partition value. Aggregations can be sum, count, min or avg. Blitz supports other operators like semi-join but we do not present them here due to space limitations.

A query Q is simply a composition of these operators applied to a single input table. For the purpose of our application, it is enough to consider single-input and single-output queries, although the techniques used in our analysis can be easily generalized to multi-input and multi-output queries as well. We sometimes write queries in a `view` form as a sequence of assignments that each apply a single operator, i.e., $v_i = \text{op}(v_j)$ or $v_i = \text{op}(v_j, v_k)$, with $i > j, i > k$, $\text{op} \in \{\sigma, \pi, :::, \bowtie, \gamma, \rho\}$. Such a query maps input table v_0 to the output table v_{max}, where v_{max} is the last variable assigned in the sequence of assignments.

Following are some examples of how SQL syntax translates to our language:

`VIEW v = <QUERY>`	$\stackrel{\text{def}}{=} v = Q$
`SELECT * FROM r;`	$\stackrel{\text{def}}{=} r$
`SELECT * FROM r WHERE phi;`	$\stackrel{\text{def}}{=} \sigma_{\text{phi}}(r)$
`SELECT A FROM r;`	$\stackrel{\text{def}}{=} \pi_A(r)$
`SELECT * FROM r1 UNION r2;`	$\stackrel{\text{def}}{=} r_1 ::: r_2$
`SELECT * FROM r1 JOIN r2 ON phi;`	$\stackrel{\text{def}}{=} r_1 \bowtie_{\text{phi}} r_2$
`SELECT * FROM r AS r1;`	$\stackrel{\text{def}}{=} \rho_{\text{cols}(r) \mapsto r1.\text{cols}(r)}(r)$
`SELECT A, F(B)FROM r GROUP BY A HAVING phi;`	$\stackrel{\text{def}}{=} \sigma_{\text{phi}}(\gamma_{A,F_B}(r))$
`SELECT A AS Aprime FROM r;`	$\stackrel{\text{def}}{=} \rho_{A \mapsto A'}(r)$

Furthermore, other queries can be desugared to fit in our language. For instance, `SELECT DISTINCT A FROM r` can be rewritten to `SELECT A FROM r GROUP BY A`.

While we use the ordering of rows in a table for implementation reasons (e.g., we may sort to reduce runtime complexity), the application views a table as a multiset in line with standard SQL semantics. Thus, we say that two queries Q_1 and Q_2 are *equivalent*, if given the same input they produce the same output up to a reordering of rows.

Groupwise queries. Consider the query in Figure 1. Each view contains either an equi-join or a group-by on the `user` column. Hence, the query can be executed as follows: (*i*) partition `wcs` on the `user` column, (*ii*) execute the query on each partition, and (*iii*) combine the partial results using a **`union`**. Such queries, where partitions of the input can be considered in isolation are called *groupwise* queries [3]; and the processing required on each partition is referred to as the *partial query*. The partial query can be the original query itself, or a simplified version based on the fact that each partition carries a unique value for the partitioning columns.

An analysis to determine if a query is groupwise (and, if so then on what columns) is described in Figure 5 in the form of inference rules. The notation $\Gamma \vdash \text{gw}(A, Q)$ means that the query Q is groupwise on columns A of its input table and Γ is a set of column renamings, mapping columns of the output table to columns of its input table. Tracking renamings is important because the set of partitioning columns of a groupwise query must refer to its input table.

The rule Init says that the identity query (which simply returns the input table) is groupwise on all columns of the table. Renaming (Rename) does not change the groupwise nature of a query but we keep of the fact that B' are aliases of columns $\Gamma(B)$ of the input table. Selection (Select) and projection (Project) don't change the groupwise nature of a query as well. In Project, the notation $\Gamma|_{A_2}$ means that we drop all mappings on columns other than A_2. For a join $Q_1 \bowtie_\varphi Q_2$, we look at the join predicate φ to identify the equi-join columns ($\{a_1, \cdots, a_n\}$) and take an intersection with the groupwise columns of Q_1 and Q_2. For a group-by query $\gamma_{A_2}(Q)$, we intersect $\Gamma(A_2)$ with the groupwise columns of Q. For a union $Q_1 ::: Q_2$, the query is groupwise on the intersection of the groupwise columns of Q_1 and Q_2.

4 UDO TEMPLATE

We use program synthesis to generate a UDO that is equivalent to a given query. Formally, given a query Q that operates over table `input`, where `input` is pre-partitioned on columns `A`, we produce a set of sort-columns `B` and a function `udo` such that `udo(sort(B, input))` is equivalent to Q(`input`). The use of column sets `A` and `B` is crucial for integrating the synthesis result with the rest of the tooling pipeline of Blitz (§5).

This section outlines a *template* that defines a space of possible combinations of sorting columns `B` and implementation `udo` that the synthesis engine will search over. The template is presented in a programmatic fashion in Figure 6 and referred to as a `super_udo` to denote the fact that it searches over the sorting columns and the `udo` at the same time. As we detail in the rest of this section, the template makes heavy use of information extracted from the input query Q to keep the synthesis task manageable. We highlight such extracted information by boxing it in the figures.

The template uses meta-operators **`repeat`** and **`choose`** that are directives to the synthesis engine: **`repeat`**`(n){B}` for a constant `n` must be replaced by at most `n` instantiations of

$$\frac{Q = r \quad r : \text{table}}{\emptyset \vdash \text{gw}(\text{cols}(r), Q)}\ \textsc{Init} \qquad \frac{\Gamma \vdash \text{gw}(A, Q)}{\Gamma \cup \{B' \mapsto \Gamma(B)\} \vdash \text{gw}(A, \rho_{B \mapsto B'}(Q))}\ \textsc{Rename}$$

$$\frac{\Gamma \vdash \text{gw}(A, Q)}{\Gamma \vdash \text{gw}(A, \sigma_{\varphi}(Q))}\ \textsc{Select} \qquad \frac{\Gamma \vdash \text{gw}(A_1, Q)}{\Gamma|_{A_2} \vdash \text{gw}(A_1, \pi_{A_2}(Q))}\ \textsc{Project}$$

$$\frac{\Gamma_1 \vdash \text{gw}(A_1, Q_1) \quad \Gamma_2 \vdash \text{gw}(A_2, Q_2) \quad \Gamma = \Gamma_1 \cup \Gamma_2 \quad \Gamma(\varphi) = \bigwedge_i^n a_i = a_i \wedge \psi \quad A_1 \cap A_2 \cap \{a_1, \ldots, a_n\} \neq \emptyset}{\Gamma \vdash \text{gw}(A_1 \cap A_2 \cap \{a_1, \ldots, a_n\}, Q_1 \bowtie_{\varphi} Q_2)}\ \textsc{Join}$$

$$\frac{\Gamma \vdash \text{gw}(A_1, Q) \quad A_1 \cap \Gamma(A_2) \neq \emptyset}{\Gamma \vdash \text{gw}(A_1 \cap \Gamma(A_2), \gamma_{A_2, F(B)}(Q))}\ \textsc{GroupBy} \qquad \frac{\Gamma_1 \vdash \text{gw}(A_1, Q_1) \quad \Gamma_2 \vdash \text{gw}(A_2, Q_2)}{\Gamma_1 \cup \Gamma_2 \vdash \text{gw}(A_1 \cap A_2, Q_1 ::: Q_2)}\ \textsc{Union}$$

Figure 5: Groupwise query analysis.

B; **choose**{S} must be replaced by one of the expressions in the set S. Note that the statements are evaluated from outermost to innermost; if a **choose**{S} statement is nested within a **repeat**(n){B} block, then first the block B is instantiated possibly multiple times, revealing multiple **choose**{S} statements, each of which are resolved independently.

The template operates on a list of records (i.e., a table). The helper functions update and predicate are defined in Figures 7 and 8, respectively. The template is structured so that any possible instantiation will run in linear time (modulo sorting) and require at most constant space in addition to storing the input and output tables. Each loop iterates over the input just once and loops are not nested.

The UDO template is chosen such that it is at least able to synthesize SQL operators such as selection, projection, and group-by. To this end the template makes use of the following program constructs.

Let the set of aggregation functions used in the input query be f_i for $1 \leq i \leq m$. Let init(f) refer to the initial value of aggregation f. For example, init(sum) is 0, whereas init(min) is ∞. The template uses a single object called scope with a fixed number of fields: Boolean fields flag_i for $1 \leq i \leq n$ (fixed constant n), integer or rational fields v_i for $1 \leq i \leq m$ (one for each aggregation function used in the input query), a list sort_cols of column names that the UDO will sort, a record old_row for memoizing the previously processed row, a Boolean flag do_break used for breaking out of loops early, and finally a list res to store the output.

The template starts by initializing the scope object (lines 3–9). Next, it assumes that the input is pre-partitioned on columns A (line 11) by assuming that all values in A are the same. In lines 13–17, the template chooses an arbitrary set of columns and sorts the input (lexicographically) on those columns. Lines 19–25 are optional loops for computing aggregations, predicates and expressions that require a single pass over all records. Lines 28–31 contain a mandatory loop that in addition to the above computes the output.

```
1  List super_udo(List input) {
2    // init flags, i in [1,n]
3    scope.flag_i = false;
4    // init aggregation, i in [1,m]
5    scope.v_i = init(f_i);
6    // init sorting columns
7    scope.sort_cols = nil;
8    // init output
9    scope.res = nil;
10   // input is pre-partitioned on columns A
11   assume all_equal(input, A);
12   // choose sorting columns
13   if(choose{true, false}) {
14    repeat(p1) {
15     scope.sort_cols.add(choose{cols(input)});
16    } input = sort(input, scope.sort_cols);
17   }
18   // optional aggregation loops
19   if(choose{true, false}) {
20     repeat(p2) {
21       scope.do_break = false;
22       foreach(row in input) {
23         if(scope.do_break) break;
24         update(scope, row, false);
25   } } }
26   // aggregation and output loop
27   scope.do_break = false;
28   foreach(row in input) {
29     if(scope.do_break) break;
30     update(scope, row, true);
31   }
32   return scope.res;
33 }
```

Figure 6: The UDO template.

The update template (Figure 7) is a sequence of p_3 guarded commands. A command updates fields of scope, such as setting flags, aggregating values, resetting them to their initial

```
void update(scope, row, can_output) {
  repeat(p3) {
    if(predicate(scope, row)) {
      repeat(p4) {
        choose {
          // set flags, i in [1,n]
          scope.flag_i = true,
          // set break
          scope.do_break = true,
          // reset aggregation, i in [1,m]
          scope.v_i = init(f_i),
          // aggregate, column a, i in [1,m]
          scope.v_i = f_i(scope.v_i, row.a),
          // memoize current row
          scope.old_row = row,
          if(can_output) {
            // add to output
            scope.res.add(cat(row, scope.v_1, ...,
                 scope.v_m))
          }
        };
} } } }
```

Figure 7: Updates scope or adds a row to output.

```
bool predicate(scope, row) {
  p = choose{true, false};
  repeat(p5) {
    t = choose {
      // check query predicate φ in Q
      normalize(φ, scope, row),
      // check flags, i in [1,n]
      scope.flag_i == true,
      // check partition boundary, a in scope.
          sort_cols
      scope.old_row.a == row.a,
      // check if last row
      is_last(row),
      true
    };
    t = choose{!t, t};
    p = choose{p && t, p || t};
  }
  return p;
}
```

Figure 8: Template for a predicate.

value, or adding a single record to the `res` field. Each command is guarded by a predicate synthesized in `predicate` (Figure 8). In our experiments, we found that often the same predicate is needed for multiple commands. Hence, the `repeat(p4)` block appears inside the predicate guard.

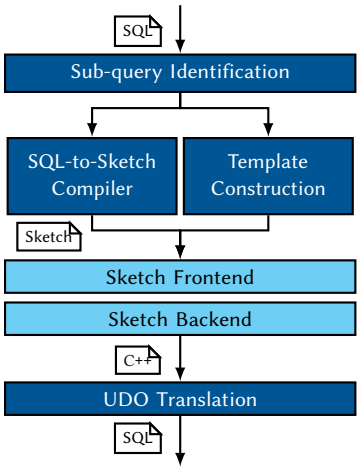

Figure 9: Blitz tool chain

The `predicate` template searches over possible predicates consisting of p_5 terms, where each term can check a flag, check a predicate that appeared in the input query, compare the `old_row` against the current, check if the current row is the last in the table, or just be the constant `true`. A term can optionally be negated (`!`) and then conjoined using either conjunction (`&&`) or disjunction (`||`).

Predicates that are extracted from the input query need to be *normalized* because they might refer to columns of intermediate tables, whereas the UDO operates on the input table. We undo renamings and replace column names with ones appearing in `row`. We also replace aggregation functions by intermediate results in the `scope` object. For example, consider the query in Figure 4. We extract the predicate `m1 > 100` from the query, but to use it, we normalize it to `scope.v_i > 100`, where `v_i` is being used for a `sum` aggregation.

Remark. A group-by, for instance, can be synthesized by first sorting over its partitioning columns, then iterating over this sorted order and using the field `old_row` to remember the previous row. When `old_row` differs from `row` on the sort columns, the UDO knows that the partition has changed; it can then reset any aggregated value and start aggregating the next partition. Of course, the UDO can do much more, for example, output multiple rows per group (unlike a group-by that outputs a single row per group) or replace the complex self-join of Figure 1 with iteration and flags (Figure 3).

5 QUERY OPTIMIZATION USING BLITZ

Blitz is a query-to-query transformation. Its workflow is shown in Figure 9. This section describes the various components of Blitz.

The first stage of Blitz identifies sub-queries of the input query that are candidates for optimization. This is done as follows. Blitz looks at the input query Q in its `view` form as a sequence of assignments and constructs the query DAG. Nodes in the DAG correspond to variables and edges correspond to operators. For instance, an assignment $v_i = op(v_j, v_k)$ results in the edges (v_j, v_i) and (v_k, v_i) getting added to the query DAG. In this DAG, for each node n that is not the result of a select, project or a rename operator, Blitz performs a groupwise analysis that starts at n and is limited to only nodes dominated by n. Once this process finishes, Blitz picks the largest sub-queries (greedily) that are groupwise on at least one column. This process is quadratic time in the number of union, join and group-by operators in the query but quite efficient in practice. Let Q_{cand} be one such sub-query. The subsequent stages of Blitz are repeated for each such Q_{cand}.

The second stage constructs the input to the synthesis tool. Blitz uses a program synthesis tool called Sketch [19]. Sketch accepts a program written in an imperative-style language (also referred to as a *sketch*). This program can contain assertions, `repeat` meta statements and it can contain *holes*, which are used in `if-else` cascades to encode choices between expressions and statements, much like the template that we described in the previous section. The goal of Sketch is to resolve all choices such that the resulting program satisfies all its assertions.

It is important to note that Sketch can only provide guarantees of correctness under a bound on the size of the input (in our case, the size of the input table). Currently, the output of Blitz has to be manually verified for correctness, but so far we found that either the synthesis fails or it produces a correct output. This limitation has a bearing on the treatment of complex predicates. For instance, a predicate $x \geq 5000$, where x is a count on the number of rows in a table, will only be satisfied by tables with at least 5000 rows. The synthesizer will not be able to satisfy the predicate (because it operates on small input tables) and learn anything about subsequent parts of the query. Blitz replaces large constants with smaller ones, performs the synthesis, and then replaces back the original values. This is possible because the UDO template borrows predicates directly from the original query (Figure 8). Blitz remembers the origin of each predicate in the template as well as the synthesized UDO.

Blitz translates Q_{cand} to a sketch program without holes. This is done via a simple compilation of relational operators to imperative code that implements the operators. Next, Q_{cand} is used to generate a sketch based on the template described in the previous section. Finally, these two sketches are put together using a harness. The harness calls these sketches one after the other to obtain their outputs (say, `spec_out` and `udo_out`) and asserts they are equivalent:

```
assert size(spec_out) == size(udo_out);
foreach (srow in spec_out) {
  bool matches = false;
  foreach (urow in udo_out)
    matches = matches || (srow == urow);
  assert matches;
}
```

The resulting sketch, along with the harness, is fed to Sketch. Blitz has a multitude of options to heuristically configure the template which helps in scaling the synthesis task. Blitz does this by fixing the constants $p_1, \ldots, p_5$ that control the `repeat` blocks in the template (Figures 6, 7 and 8). Blitz additionally makes use of a parameter (let's call it p_6) to decide whether to split the query predicates φ into their sub-terms when constructing the template. The three configurations that we use in our experiments are $(p_1, p_2, p_3, p_4, p_5, p_6) = \{(1, 1, 3, 2, 3, \mathsf{F}), (2, 1, 5, 1, 3, \mathsf{F}), (2, 1, 5, 2, 4, \mathsf{T})\}$. The number of Boolean flags (`flag_i`) was fixed at 2. Blitz then spawns three Sketch instances–one for each configuration–in parallel and picks the solution of the fastest instance. Typically, the fastest solution comes from a run with smaller parameters when it exists, resulting also in a simpler UDO. Sketch outputs, by default, a C++ program when the synthesis succeeds. Finally, Blitz extracts the UDO implementation and the sorting columns by inspecting the C++ program and translating it to the input language expected by the big-data platform that the original query was targeting. Currently, this translation is done manually but the process is mechanical and amenable to automation.

6 QUERY ANALYSES

This section describes further query analyses used by Blitz for scaling the synthesis process. These are used to split a query into smaller ones to deal with N-ary unions and joins (§6.1), perform taint analysis to simplify the UDO template (§6.2) and identify redundant columns that can be eliminated altogether from the query (§6.3).

6.1 Query splitting

Operators like join and union combine multiple data inputs. When done in succession to combine more than two data sources, e.g., $Q_1 \bowtie (Q_2 \bowtie Q_3)$, we refer to them as N-ary operators. Blitz identifies N-way joins and unions and tries to split the query into $N + 1$ queries; the first N each compute one of the inputs to the N-ary operator, and the last one actually runs the operator. For example, the query in Figure 4 can be broken into six queries: one each to compute tables `Out1` to `Out5` and the last one that does a union of these five tables. Each smaller query is now much simpler to optimize. However, the challenge is to put the synthesized UDOs

together in a way that is efficient (with linear complexity) while preserving semantics.

Composing UDOs. Consider a query Q that is either a join or a union of N sub-queries Q_1 to Q_N. If Blitz has identified this query as a target for optimization, then Q must be groupwise. Let A be the groupwise columns of Q. Going by Figure 5, it must be the case that queries Q_i are also groupwise on A (although they may be groupwise on more columns). Blitz invokes the synthesizer on each Q_i while enforcing A as the groupwise columns. (If any of the synthesis task fails then Blitz gives up trying to optimize Q.) Suppose it obtains the pair (udo_i, s_i) of the UDO and sorting order for Q_i. Blitz checks if each s_i is a prefix of a common list s. If this holds, then s will be the sorting order of the composed UDO. Next, Blitz goes on to determine if the UDOs can be fused into a single linear-complexity UDO.

Union. In the case of union, fusion is always possible.[3] We can simply call the individual UDOs one after the other on different `scope` objects, except that they use the same `res` field for output. A more efficient way is to fuse the update loops of the individual UDOs into one loop and fuse the output loops into a single loop as well. Figure 10a shows the merged UDO that combines UDOs of `Out1` and `Out2` of Query2 (Figure 4). The colors indicate parts of the individual UDOs.

Join. For join, fusion need not always be possible. However, groupwise queries provide an interesting opportunity for linear implementations of (equi-)joins. Note that because the input query Q is groupwise on A, then by the Join rule of Figure 5, the N-ary join must be an equi-join on columns A. If each of the N UDOs produce a single output per partition on A (in general, we only require $N-1$ UDOs to satisfy this condition) then we can combine them into a single linear-time UDO. This condition needs to be checked; for example, the query from Figure 1 does not satisfy this requirement. If the condition is satisfied, the UDOs are fused together (refer Figure 10b), just like for union, but instead of adding rows to the `res` field, the UDO concatenates rows together to simulate the join operation.

Ensuring composability. Blitz does not perform synthesis first and then later check if the resulting UDOs and sorting orders can be put together. Instead, it adds additional constraints to the synthesis tasks so that if each of the N synthesis tasks succeed then the composition is always possible. For the sorting order, it first chooses a single sorting order s, then it enforces that each synthesis task only consider prefixes of s. Further, in the case of N-ary joins, it restricts the UDO templates to produce output just once.

[3]Note that in our language a union produces a multiset and does not eliminate duplicates. This condition does not apply to a *distinct* union.

```
proc udo2_mergedU(a, List rows)
  // assumes sorted([c], rows)
  first ← True
  foreach (c,m1,m2) ∈ rows
    if first
      sum1 ← 0; cnt1 ← 0;
      sum2 ← 0; cnt2 ← 0;
      oldC ← c; first ← False;
    if oldC == c
      sum1 ← sum1 + m1
      sum2 ← sum2 + m2
    else
      if (sum1 > 100) cnt1++;
      if (sum2 > 100) cnt2++;
      sum1 ← m1;sum2 ← m2;
      oldC ← c;
  if (sum1 > 100) cnt1++;
  if (sum2 > 100) cnt2++;
  if cnt1 > 0
    output(a, cnt1)
  if cnt2 > 0
    output(a, cnt2)
```

(a) UNION: `Out1` ::: `Out2`

```
proc udo2_mergedJ(a, List rows)
  // assumes sorted([c], rows)
  first ← True
  foreach (c,m1,m2) ∈ rows
    if first
      sum1 ← 0; cnt1 ← 0;
      sum2 ← 0; cnt2 ← 0;
      oldC ← c; first ← False;
    if oldC == c
      sum1 ← sum1 + m1
      sum2 ← sum2 + m2
    else
      if (sum1 > 100) cnt1++;
      if (sum2 > 100) cnt2++;
      sum1 ← m1;sum2 ← m2;
      oldC ← c;
  if (sum1 > 100) cnt1++;
  if (sum2 > 100) cnt2++;
  if cnt1 > 0 ∧ cnt2 > 0
    output(a, cnt1, cnt2)
```

(b) JOIN: `Out1` $\bowtie_{\varphi a}$ `Out2`

Figure 10: Merged UDOs for union (left) and join (right).

6.2 Taint analysis

We designed a *query taint analysis* to determine which columns of the input table of a query can influence what columns in its output. Blitz uses taint relationships in various ways. First, it eliminates input columns that do not taint any of the output columns. For example, in Figure 4, columns `m2` to `m6` do not influence any of the columns of the `Out1` table. Second, Blitz identifies *flow-through* columns, which are input columns that do not influence any other column, i.e., they simply flow to the output unmodified. The `sales` column of Figure 1 is one such example. Flow-through columns can be excluded for consideration in the UDO template in all places, except when output is added to `scope.res`. That is, they need not be considered for sorting, or in a predicate or aggregation.

The taint analysis is presented formally in Figure 11 as inference rules. A judgement $\Gamma \vdash \text{ta}(R, Q)$ means that the query Q carries the taint relationship R, where R is a binary relation from input columns to output columns, i.e., if $(a, b) \in R$ then input column a potentially influences output column b. As in the groupwise analysis, we use Γ to carry column renaming information.

$$\frac{Q = r \quad r : \text{table}}{\emptyset \vdash \text{ta}(\{(a,a) \mid a \in \text{cols}(r)\}, Q)}\ \textsc{Init} \qquad \frac{\Gamma_1 \vdash \text{ta}(R, Q) \quad \Gamma = \Gamma_1 \cup \{B' \mapsto \Gamma_1(B)\}}{\Gamma \vdash \text{ta}(\{(b, \Gamma(a)) \mid (b,a) \in R\}, \rho_{B \mapsto B'}(Q))}\ \textsc{Rename}$$

$$\frac{\Gamma \vdash \text{ta}(R, Q)}{\Gamma \vdash \text{ta}(R \cup \text{cols}(\Gamma(\varphi)) \times \text{cols}(\sigma_\varphi(Q))), \sigma_\varphi(Q))}\ \textsc{Select} \qquad \frac{\Gamma \vdash \text{ta}(R, Q)}{\Gamma|_A \vdash \text{ta}(\{(b,a) \mid (b,a) \in R, a \in A\}, \pi_A(Q))}\ \textsc{Project}$$

$$\frac{\Gamma_1 \vdash \text{ta}(R_1, Q_1) \quad \Gamma_2 \vdash \text{ta}(R_2, Q_2) \quad \Gamma = \Gamma_1 \cup \Gamma_2}{\Gamma \vdash \text{ta}(R_1 \cup R_2 \cup (\text{cols}(\Gamma(\varphi)) \times \text{cols}(Q_1 \bowtie_\varphi Q_2))), Q_1 \bowtie_\varphi Q_2)}\ \textsc{Join} \qquad \frac{\Gamma_1 \vdash \text{ta}(R_1, Q_1) \quad \Gamma_2 \vdash \text{ta}(R_2, Q_2)}{\Gamma_1 \cup \Gamma_2 \vdash \text{ta}(R_1 \cup R_2, Q_1 ::: Q_2)}\ \textsc{Union}$$

$$\frac{\Gamma \vdash \text{ta}(R, Q)}{\Gamma \vdash \text{ta}(R \cup (\Gamma(A) \times \text{cols}(\gamma_{A,F_B}(Q))), \gamma_{A,F_B}(Q))}\ \textsc{GroupBy}$$

Figure 11: Taint analysis.

$$\frac{Q = r \quad r : \text{table}}{\emptyset \vdash \text{rd}(\mathit{Equiv}(\text{cols}(r)), Q)}\ \textsc{Init} \qquad \frac{\Gamma \vdash \text{rd}(E, Q)}{\Gamma \cup \{B' \mapsto \Gamma(B)\} \vdash \text{rd}(E, \rho_{B \mapsto B'}(Q))}\ \textsc{Rename}$$

$$\frac{\Gamma \vdash \text{rd}(E, Q)}{\Gamma \vdash \text{rd}(E - \text{cols}(\Gamma(\varphi)), \sigma_\varphi(Q))}\ \textsc{Select} \qquad \frac{\Gamma \vdash \text{rd}(E, Q)}{\Gamma|_A \vdash \text{rd}(E \sqcap \mathit{Equiv}(\Gamma(A)), \pi_A(Q))}\ \textsc{Project}$$

$$\frac{\Gamma_1 \vdash \text{rd}(E_1, Q_1) \quad \Gamma_2 \vdash \text{rd}(E_2, Q_2) \quad \Gamma = \Gamma_1 \cup \Gamma_2 \quad \Gamma(\varphi) = \bigwedge_i^n a_i = a_i \wedge \psi}{\Gamma \vdash \text{rd}((E_1 \sqcap E_2 \sqcap \mathit{Equiv}(\{a_1, \ldots, a_n\})) - \text{cols}(\psi), Q_1 \bowtie_\varphi Q_2)}\ \textsc{Join}$$

$$\frac{\Gamma \vdash \text{rd}(E, Q)}{\Gamma \vdash \text{rd}((E \sqcap \mathit{Equiv}(\Gamma(A))) - \Gamma(B), \gamma_{A,F_B}(Q))}\ \textsc{GroupBy} \qquad \frac{\Gamma_1 \vdash \text{rd}(E_1, Q_1) \quad \Gamma_2 \vdash \text{rd}(E_2, Q_2)}{\Gamma_1 \cup \Gamma_2 \vdash \text{rd}(E_1 \sqcap E_2, Q_1 ::: Q_2)}\ \textsc{Union}$$

Figure 12: Redundant column analysis.

The analysis starts with the identity relationship (rule Init) for the empty query. The rules for renaming and projection are straightforward. Whenever the analysis encounters a predicate (in Select or Join), it adds a taint from all columns that appear in the predicate to all output columns. For a group-by, we add a taint from all partitioning columns to all output columns.

6.3 Redundant column analysis

We define a set of columns to be *redundant* if they always occur together in the partitioning columns of a group-by or an equi-join operator and never in a filtering predicate. Essentially, redundant columns are treated in a similar manner by the query. They can be replaced by a single column, which carries a tuple of values of the redundant columns. Blitz, for simplicity, instead drops all columns but one from a redundant set of columns. This offers reduction in the complexity of the UDO template. Once the UDO is synthesized, the redundant columns are reintroduced. For `Out1` in Figure 4, {`a`, `b`} forms a redundant set.

The analysis is presented formally in Figure 12, following a similar style as the other analyses presented in this paper. The judgement $\Gamma \vdash \text{rd}(E, Q)$ defines an equivalence relation E over the set of input columns of Q. The equivalence classes of E define all redundant column sets of the query. The figure requires extra notation. For a set A of columns, $\mathit{Equiv}(A)$ is the equivalence class where $(a, b) \in \mathit{Equiv}(A)$ if and only if $a \in A$ and $b \in A$. For two equivalence relations E_1 and E_2, $E_1 \sqcap E_2 = \bigcup_{\mathit{Equiv}(A_i) \in E_1, \mathit{Equiv}(B_j) \in E_2} \{\mathit{Equiv}(A_i \cap B_j), \mathit{Equiv}(A_i - \text{cols}(E_2)), \mathit{Equiv}(B_j - \text{cols}(E_1))\}$ is their partition. Finally, $E - A$ is the same as the equivalence E except that columns in A are removed from all equivalence classes, i.e., $E - A = \bigcup_{\mathit{Equiv}(B) \in E} \mathit{Equiv}(B - A)$.

For example, consider the following query (which is a simplified version of `Q3` from our benchmark suite). It operates on a table r with columns $\{a, \ldots, e\}$.

$$\begin{aligned} v_1 &= \sigma_{e>0}(\gamma_{\{a,b,c,d\},\text{sum}(e)}(r)) \\ v_2 &= \rho_{\{a,b,e\} \mapsto \{a',b',e'\}}(\gamma_{\{a,b\},\text{sum(e)}}(v_1)) \\ v_3 &= v_1 \bowtie_{a=a' \wedge b=b'} v_2 \end{aligned}$$

The analysis of Figure 12 will output that columns a and b are equivalent, and so are columns c and d. Thus, we drop columns b and d and the resulting query will be the following, which operates on a table r with columns $\{a, c, e\}$:

$$\begin{aligned} v_1 &= \sigma_{e>0}(\gamma_{\{a,c\},\text{sum}(e)}(r)) \\ v_2 &= \rho_{\{a,e\} \mapsto \{a',e'\}}(\gamma_{\{a\},\text{sum(e)}}(v_1)) \\ v_3 &= v_1 \bowtie_{a=a'} v_2 \end{aligned}$$

Table 1: This table shows benchmark queries and number of stages in their query plans from two big-data engines. It also shows the number of stages in the optimizable sub-query identified by Blitz and a count of the important operators (Group By γ, Join $\bowtie$, Union $\cup$) in the sub-query.

	Spark		Scope		sub-query
	stages	sub-query	stages	sub-query	operators
Q1	6	6	10	8	1 γ,2 $\bowtie$
Q2	16	16	13	13	5 γ,4 $\cup$
Q3	8	6	10	7	2 γ,1 $\bowtie$
Q4	12	9	9	8	4 γ,2 $\cup$
Q5	16	16	7	7	5 γ,4 $\bowtie$
Q6	5	3	4	3	2 γ
Q7	10	8	12	9	3 γ,2 $\bowtie$
Q8	7	7	8	8	4 γ,2 $\cup$
Q9	15	15	12	12	7 γ,6 $\bowtie$
Q10	11	9	20	18	8 γ,7 $\cup$
Q11	4	2	11	9	3 γ,2 $\cup$
Q12	9	7	14	11	2 γ,2 $\bowtie$
Q13	5	5	7	7	2 γ,1 $\bowtie$
Q14	6	4	5	4	1 γ,2 $\cup$
Q15	4	2	11	9	1 γ,1 $\bowtie$
Q16	4	2	4	2	1 γ,1 $\bowtie$
Q17	6	4	5	4	2 γ, 1 $\bowtie$
Q18	5	4	9	7	1 γ,1 $\bowtie$
Q19	5	3	4	3	2 γ,1 $\bowtie$
Q20	4	4	6	6	2 γ,1 $\bowtie$

7 EVALUATION

We evaluated Blitz on a set of production queries from Microsoft. We first describe our benchmarks.

7.1 Benchmark selection

We analyzed a log of all Scope jobs executed on one of Microsoft's clusters (with more than 50,000 nodes) in a single day. The log included query scripts, their query plans as generated by Scope and execution time statistics. It had about 90,000 queries that cumulatively took nearly 790,000 minutes to execute. Many of the queries executed repeatedly [15] and some of them were short running (<30 minutes). We post-processed the log and found there were about 1,100 unique long-running jobs. They cumulatively accounted for 80% of the total execution time. Among these, we found 375 queries (34%) to have at least one stage that dominated three or more stages.

Scope has a rich programming interface [28]. It allows the use of arbitrary .NET types and functions from within the query. Our current implementation, based on the language described in Section 3, does not support .NET types and functions.[4] Given a query, we extracted its largest dominated sub-query (i.e., the sub-query corresponding to the stages dominated by the most dominant stage) and analyzed it for the absence of .NET types, functions and other unsupported Scope features. Of the 375 queries, we found that Blitz can currently support 148 queries, which cumulatively take 92,250 minutes of execution time. Among these, we randomly picked 20 queries to form our benchmark set.

The input data of these queries was customer proprietary, so we substituted it with data from TPCDS [22], a standard database benchmark suite. (This also allowed us to evaluate our optimization on Spark.) While generating the input data, we made sure to match the sizes of intermediate outputs of each stage, based on the available runtime statistics. Performing this matching was difficult for queries that used many input tables; we had to discard a query (and pick another one) if it used more than three input tables.

Table 1 shows the high-level characteristics of our benchmark queries. The table also contains statistics about the query plans generated by Scope and Spark; it shows the number of stages in the plan[5] and how many of them were included in a sub-query picked by Blitz for optimization. As the table shows, a significant fraction (sometimes all) of the stages are marked for optimization. Finally the table reports the number of group-by, join and union operators in the sub-query. The sub-queries are quite complex, some with more than 5 joins or unions. But, as we will see in the next few sections, Blitz is still able to scale to them.

7.2 Experimental setup

We ran the optimized queries returned by Blitz on both Spark and Scope on two separate clusters. One was a production Microsoft cluster running Scope and the other was a 16-node standalone Spark (ver. 2.0.2) cluster running on Azure (D4 v2 VMs). The production cluster had a little more than 50,000 nodes and our jobs were restricted to use at-most 1,000 tokens [2] (each token is a bundle of 2 cores and 8GB of RAM). Blitz itself was run on a standard desktop machine (AMD Ryzen 1700 3.0GHz CPU with 8 cores, 16GB RAM).

7.3 Synthesis results

Results from running Blitz are reported in Table 2. Blitz ran the synthesis engine with a time limit of 10 min. The synthesis succeeded in all cases except for one (more on `Q20` in §7.5). In fact, synthesis finished for 14 queries within a minute. The table also shows which analyses were useful

[4] We lower SQL types such as `float` and `double` to `int`; and we lower a `string`-typed column to `int` provided equality is the only operator that is applied to the column, i.e., it only participates in a join or group-by.

[5] The query plans generated by Scope and Spark differ because of differences in the query optimization process employed by the two.

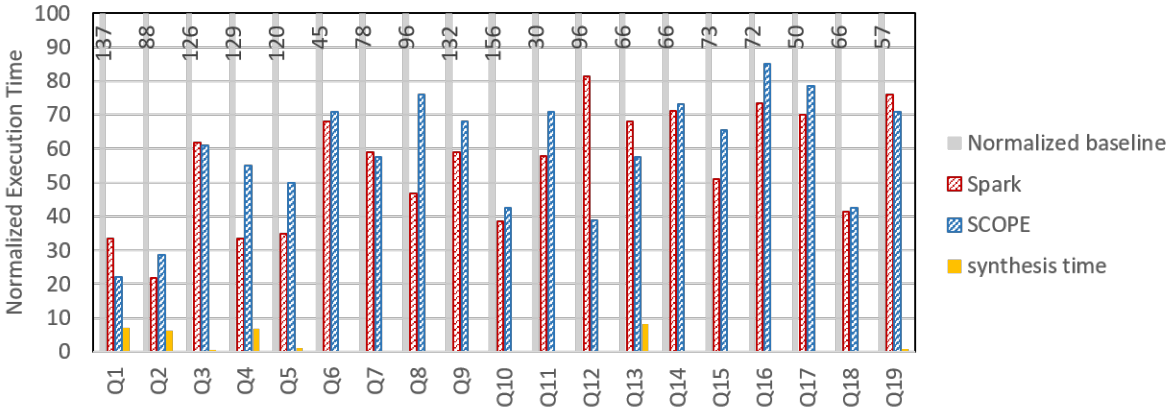

Figure 13: Performance comparison on Spark and Scope. The figure shows the baseline execution time (as labels) and normalized execution times on Spark (normalized to baseline execution on Spark) and Scope (normalized to baseline execution on Scope). All running times are in minutes.

Table 2: Results from running Blitz. Column "simpl. anal." refers to analyses required: s (query splitting, §6.1) or t (taint analysis, §6.2), or r (redundancy analysis, §6.3).

	Synth.	Simpl.	UDO complexity		Sort	LOC
	time(s)	anal.	time	space	cols	
Q1	516	t	1 loop	constant (~5)	1	20
Q2	221	s,t,r	1 loop	constant (~30)	1	60
Q3	44	s,r	2 loops	linear (n+~20)	2	60
Q4	460	s,r	1 loop	constant (~10)	2	50
Q5	79	s	1 loop	constant (~20)	1	50
Q6	30	r,t	2 loops	linear (n+~10)	0	20
Q7	18	s	2 loops	linear (n+~10)	1	50
Q8	2	s	1 loop	constant (~10)	1	50
Q9	2	s,t	1 loop	constant (~20)	1	80
Q10	3	s	1 loop	constant (~20)	0	80
Q11	3	s,t	2 loops	linear (n+~20)	0	90
Q12	14	s	2 loops	linear (n+~10)	1	40
Q13	330	-	2 loops	linear (n+~10)	2	40
Q14	1	s	1 loop	constant (~10)	0	30
Q15	2	-	1 loop	constant (~10)	1	10
Q16	5	s	1 loop	constant (~10)	1	10
Q17	13	s	2 loops	linear (n+~20)	1	60
Q18	1	t	1 loop	constant (~5)	1	50
Q19	31	r	1 loop	constant(~10)	0	40
Q20	fail	-	N/A	N/A	N/A	N/A

(column "simplifying analysis") for each query. In particular, query splitting leads to significant gains; it applies to 13 queries and simplified all N-ary queries with $N > 2$. Without this simplification, none of the queries with $N > 2$ would synthesize within the time limit. Redundant column analysis applies to fewer queries but speeds up synthesis significantly for these. Four of the five queries (all but Q19) time-out without this analysis. The rest of the columns in the table show characteristics of the synthesized UDO: time complexity, space complexity, number of sorting columns, and the lines of code (of the Python implementation of the UDO required for Spark, which we manually read out from Sketch's output). A majority of the UDOs have constant space complexity, i.e., they only require a single pass over the input rows, thus, can be executed in a streaming fashion.

7.4 Performance comparison

Figure 13 shows results from running the benchmark queries on Scope and Spark. The figure compares the normalized execution time on each of the systems. Each benchmark has four bars. The first bar is the baseline unoptimized query running time normalized to 100%. For reference, this bar is labelled with the actual execution time in minutes on Scope. The second bar shows the percentage of time needed to run the query when optimized by Blitz on Spark (compared to the baseline execution on Spark). The third bar shows the same for Scope and the last bar is the synthesis time. Blitz-optimized queries perform consistently better on both systems. They run 1.3×—4.7× faster with a mean speedup of 1.92× on Scope and 2× on Spark.

The performance improvements are largely influenced by two factors: the number of stages eliminated and the complexity of the synthesized UDO. Queries Q1, Q2, Q4, Q5, Q10 all use streaming UDOs (linear time, constant space) with 6 or more stages eliminated, leading to 2.5× or more performance improvement. Queries Q3, Q13, Q17 use linear time UDOs, and Q16, Q19 have a small number of stages eliminated. These queries still speed up in the range 1.3×—2×. Q15 is very similar to Q16, but the UDO filters out a significant

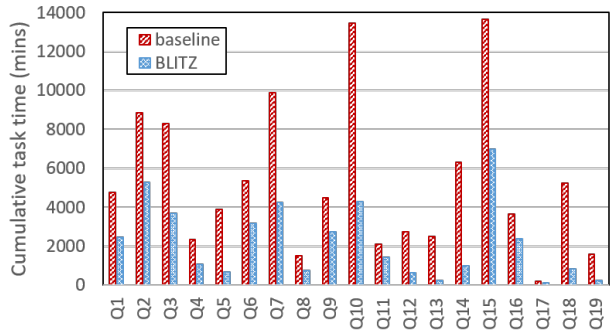

Figure 14: Cumulative time spent (in minutes) in executing benchmark queries with and without Blitz.

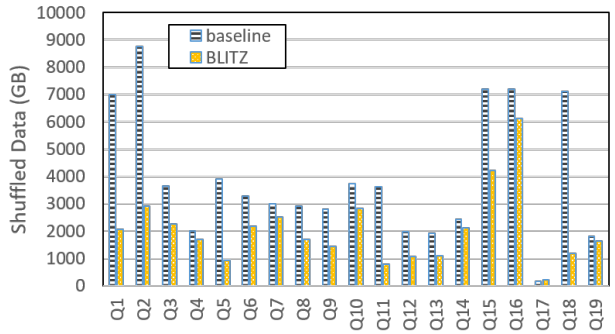

Figure 15: Total data shuffled (in GB) by the benchmark queries with and without Blitz.

amount of data, saving on computing large intermediate data, leading to bigger gains.

There are differences in the amount of improvement obtained on Spark versus that obtained on Scope. For example, with Q8, Blitz does much better on Spark than on Scope. We inspected the query plans and found out that these differences are mainly due to how the two engines read source tables and how they optimize N-ary operators. For example, when it comes to reading input tables, Spark, on one hand reads the same table multiple times (once per reference). This allows it to partition or sort the table differently for each access, if required and is also less prone to being bottlenecked on a single stage. Scope, on the other hand, introduces a single stage to read the table and shuffles only the necessary data down to subsequent stages. Both these solutions lead to unnecessary data reads and/or shuffles. Blitz's optimization eliminates the source of the problem. It avoids having to redundantly read data from a source altogether by replacing the dependant stages with a single linear-time UDO.

Resource savings. In addition to reducing execution time, Blitz also reduces the resource requirements of the query. Figure 14 shows the cumulative CPU time (defined as the sum of the times spent by all tasks used to run the query) and Figure 15 shows the total data shuffled, with and without using Blitz. Note that unlike the execution time, which is sensitive to the number of concurrent containers or VMs that the cluster can support, these metrics are a more robust measure of the quality of an execution plan. Blitz brings down the resource consumption drastically. It saves more than 50% of cumulative CPU time for all but one query, with a peak saving of close to 90%. Blitz also brings down the amount of data shuffled by up to 75%.

7.5 About Q20

We manually inspected `Q20` and found that it cannot be rewritten using a linear-time UDO. It performs two group-by operations on the input table and joins their results. It is groupwise on a single column, say a, but the group-by operators were on more than one column, say a, b and a, c, hence they produced multiple rows per partition on a. Thus, there is no way to avoid a quadratic-time loop inside the UDO to do the join.

Nonetheless, we went ahead and manually wrote the (quadratic-time) UDO and created the optimized query. Its resulting query plan was shorter than the original query because both group-by operators and the join operator were replaced by the UDO. However, its execution was slower than the original query by a factor of 2 to 3 on either of the two clusters. The original plan employed an efficient join algorithm (a sort-merge join) that is much more efficient than the nested loops inside our UDO. This justifies our requirement for insisting on linear-complexity UDOs.

8 DISCUSSION

Blitz proposes a novel approach towards optimizing big-data queries. Unlike standard query optimizers that transform SQL operators in a rule-based manner, Blitz synthesizes new UDOs directly from a SQL specification. We demonstrated that such an optimization strategy can uncover new rewrites that bring about significant performance gains. A key challenge with this methodology, however, is that our use of program synthesis does not guarantee full functional correctness. Thus, manual verification is required to ensure that the synthesized UDOs are indeed equivalent to the original query. Below we discuss why such an approach is still useful and how it can be improved in the future.

First, our experience with *bounded* synthesis has been positive. We found that the synthesis made no mistakes on the queries that we tried. Synthesis either failed completely (like in the case of `Q20`) or produced a correct result. We believe this is because the programs that we are trying to synthesize are fairly small. The synthesized UDOs contain at most 2

loops and roughly 10 statements per loop (see §5), with a total of less than 100 lines of imperative code. Although bounded synthesis does not provide full-correctness guarantees, small bounds are often sufficient to find counterexamples in such a setting and steer the synthesis to a correct solution. Further, recall that Blitz employs query simplifications (§6) before synthesis. These simplifications are sound. They not only help in speeding up the synthesis task, but they also limit the size of UDOs, thus further improving the chances of finding counterexamples within small bounds. A more thorough evaluation is required to test the limits of bounded synthesis in this domain and we plan to do this in future work.

Proving the equivalence of SQL-like programs, which has the potential of automatically verifying the results of Blitz, has gotten recent attention [6, 7, 10]. While these results are encouraging, the class of programs that Blitz works with is much larger and these results are not yet applicable. In particular, Blitz supports more operators (like GroupBy) and the target language (with UDOs) has imperative code. Further work is required to scale sound (unbounded) synthesis enough for use with Blitz.

Second, even with manual verification, Blitz adds significant value for end users of a big-data system. Over the past decade, users have mostly migrated from using low-level map-reduce abstractions to declarative querying abstractions where they rely on the query engine to figure out the most cost-effective way to execute their jobs. However, this methodology does not provide any insight when the query engine fails to optimize the query execution. Blitz identifies an important class of such scenarios, automatically constructs a (potential) optimization and escalates it to the user. Blitz emits roughly 100 lines of code within minutes, much faster than a usual programmer. The user can then verify the code returned by Blitz, instead of eagerly fine-combing all their queries for optimizations. Moreover, the manual effort is amortized in the long run because many production jobs run repeatedly (around 60% with Scope [15]). Further work is required to quantify the manual effort required to verify Blitz's proposed optimizations.

9 RELATED WORK

Classical query optimization uses equivalence rules to rewrite queries at the logical level. Once a query plan is fixed, off-the-shelf physical implementations are chosen for each logical operator. This style of query optimization has been extensively researched and is today employed in modern big-data query optimizers [1, 13, 20, 29]. While such a two-step approach has its advantages, it sometimes fails to find the best implementation of a query [3, 4, 8, 14, 27, 29, 30] especially in parallel and distributed settings. Attempts to address this include adding more rules to the optimizer [8, 14, 27, 29, 30] or extending the language [16] or at the extreme letting the end-user manually write parts of the query with user-defined operators. Rewrite rules are typically highly specialized and have low applicability. Manually writing queries or introducing UDOs requires significantly higher expertise and breaks the declarative programming abstraction. This paper proposes an alternative optimization strategy that addresses all of the above issues. It employs advanced program reasoning techniques to synthesize new query-specific operations on-the-fly. It discovers new operators that are not readily available in SQL and cannot typically be constructed in a rule-based manner. To our knowledge, our use of program synthesis for query optimization is novel.

The last decade has seen many applications of program synthesis—a technique that was considered hard to scale. There is one line of work that starts with a problem description given in terms of input-output examples. The goal is then to come up with a program that satisfies these examples. In such situations, because the specification is so under-specified, it is easy to come up with *some* program that does the job, however, the difficult part is to come up with the *right* program that a human would have written. One of the most popular examples is Microsoft Excel's FlashFill feature [11]. More related to our work are SQLSynthesize [26], Scythe [24], and BigLambda [18] which construct a SQL query or a map-reduce program from input-output examples. However, the goals of their work are very different from Blitz because the intent is to help a novice user write a declarative query. Blitz, on the other hand, already has the declarative query available and its job is to optimize the execution of the query.

Another line of work in program synthesis starts with a more detailed specification than just input-output examples. For example, Sketch [19] falls in this category, originally used to synthesize tricky parts of bit-manipulating programs and data-structure operations. Cheung et al. [5] used program synthesis to extract a SQL query out of imperative code that used low-level accesses to a database. The target of their work (SQL) is different from ours (a UDO).

Finally we note that optimizing queries with UDOs has its own challenges and newer techniques have been proposed recently [9, 12, 17] to address them. This line of work is orthogonal and all these optimizations apply to our system as well.

ACKNOWLEDGMENTS

This work was supported by the Austrian Science Fund (FWF) through grants S11403-N23 (https://arise.or.at) and W1255-N23 (http://logic-cs.at). We thank the anonymous reviewers, and our shepherd Emery Berger for their useful feedback.

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[2] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 285–300. http://dl.acm.org/citation.cfm?id=2685048.2685071

[3] Damianos Chatziantoniou and Kenneth A. Ross. 1997. Groupwise Processing of Relational Queries. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–485. http://dl.acm.org/citation.cfm?id=645923.671003

[4] Damianos Chatziantoniou and Kenneth A. Ross. 2007. Partitioned Optimization of Complex Queries. *Inf. Syst.* 32, 2 (April 2007), 248–282. https://doi.org/10.1016/j.is.2005.09.003

[5] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

[6] 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 2017)*. ACM, New York, NY, USA, 510–524. https://doi.org/10.1145/3062341.3062348

[7] Przemysław Daca, Thomas A. Henzinger, and Andrey Kupriyanov. 2016. *Array Folds Logic*. 230–248. https://doi.org/10.1007/978-3-319-41540-6_13

[8] César Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 571–581. https://doi.org/10.1145/375663.375748

[9] Diego Garbervetsky, Zvonimir Pavlinovic, Michael Barnett, Madanlal Musuvathi, Todd Mytkowicz, and Edgardo Zoppi. 2017. Static Analysis for Optimizing Big Data Queries. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 932–937. https://doi.org/10.1145/3106237.3117774

[10] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. *Verifying Equivalence of Spark Programs*. Springer International Publishing, Cham, 282–300. https://doi.org/10.1007/978-3-319-63390-9_15

[11] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

[12] 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 *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 121–133. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/guo

[13] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[14] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1235–1246. https://doi.org/10.1145/2588555.2595630

[15] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 117–134. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi

[16] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proc. VLDB Endow.* 8, 10, 1058–1069. https://doi.org/10.14778/2794367.2794375

[17] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/2815400.2815418

[18] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 326–340. https://doi.org/10.1145/2908080.2908102

[19] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 404–415.

[20] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1626–1629. https://doi.org/10.14778/1687553.1687609

[21] TPC. 2016. TPCx-BB Benchmark. (2016). http://www.tpc.org/tpcx-bb/

[22] TPC. 2017. TPC-DS Benchmark. (2017). http://www.tpc.org/tpcds/

[23] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[24] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[25] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[26] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE Computer Society, Washington, DC, USA, 224–234. https://doi.org/10.1109/ASE.2013.6693082

[27] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced Partitioning Techniques for Massively Distributed Computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2213836.2213839

[28] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[29] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[30] Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. 2003. WinMagic: Subquery Elimination Using Window Aggregation. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 652–656. https://doi.org/10.1145/872757.872840