SQL Engines Excel at the Execution of Imperative Programs

Tim Fischer Denis Hirn Torsten Grust
[tim.fischer,denis.hirn,torsten.grust]@uni-tuebingen.de
University of Tübingen, Germany

ABSTRACT

SQL query engines can act as efficient runtime environments for the execution of imperative programs over database-resident tabular data. To make this point, we lay out the details of a compilation strategy that maps the basic blocks of arbitrarily branching and looping control flow graphs into plain—possibly recursive—SQL:1999 common table expressions. The compiler does not stumble when faced with imperative programs of several hundred lines and emits SQL code that can execute such programs over entire batches of input arguments. These batches create opportunities for parallel program evaluation which contemporary query decorrelation techniques exploit automatically. SQL engines that already support UDFs may find the present program execution approach to outperform their native implementation—SQL engines without such support may gain UDF capabilities without the need to build a dedicated interpreter.

PVLDB Reference Format:
Tim Fischer Denis Hirn Torsten Grust. SQL Engines Excel at the Execution of Imperative Programs. PVLDB, 17(13): 4696 - 4708, 2024.
doi:10.14778/3704965.3704976

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at `https://github.com/flummi-compiler/PVLDBv17`.

1 A SQL ENGINE RUNS IMPERATIVE CODE

When the data resides in persistent relational tables, computation over this data should happen inside the database kernel, too ("*move your computation close to the data*" [38]). Holding on to this decades-old wisdom can be challenging if the computation is expressed in *imperative-style algorithms*, *i.e.*, in terms of statement sequencing, destructive variable updates, as well as branching and looping control flow—all of which have no obvious equivalent in a relational SQL query.

Figure 1 displays an example of such imperative-style code: Jarvis' algorithm *giftwrap*(S) iteratively determines the convex hull of a set S of two or more points in the 2D plane [25]. In this textbook-style pseudo code, operations concerned with the immediate access to the point data are enclosed in boxes . Arguably, however, the essence of Jarvis' algorithm lies *outside* these boxes, expressed through assignments to variables p_0, p that keep track of points on the convex hull, and the **repeat-until** loop that exits when the hull has been closed.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704976

Can we avoid to rely on database-external interpreters and instead let a SQL-based database engine itself efficiently execute imperative code like the above?

Imperative computation over relational data. The following pages introduce *Flummi* (German for *bouncing ball*), a compilation strategy that *translates imperative-style computation over database-resident tabular data into plain SQL queries*.

Figure 2 recasts Jarvis' algorithm of Figure 1 in the syntax of an imperative language. We consider stateful, iterative programs following this style to be typical Flummi input. In this program, data accesses in assume a table `points(cloud,label,x,y)` in which a row (c, ℓ, x, y) encodes a point labelled ℓ at coordinates (x, y); point set S is partitioned into multiple point clouds identified by c. Flummi itself is *not* concerned with data access, treating the as parameterized black boxes. Instead, our focus is on the compilation of a generic "lowest common denominator" language for imperative computation, featuring stateful variables and—in particular—arbitrary iterative control flow. Flummi consumes

giftwrap(S):
$p_0 \leftarrow$ leftmost point in point set S
$p \leftarrow p_0$
repeat
  output p (p is on the convex hull)
  $p \leftarrow$ point $p_1 \neq p$ in S such that there is no point to the left of line $p \rightarrow p_1$
until $p_0 = p$

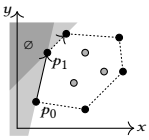


Figure 1: Finding the convex hull of point set S (with $|S| \geqslant 2$), textbook-style (Jarvis' "gift wrapping" algorithm [25]).

```
1  giftwrap(cloud) {
2          p0 <- SELECT p                                  e_A1[cloud];
                 FROM   points AS p
                 WHERE  p.cloud = $1
                 ORDER BY p.x
                 LIMIT 1
3          p  <- $1                                        e_A2[p0];
4  repeat: EMIT $1.label                                   e_E1[p]; -- p is on the convex hull
5          p <- SELECT p1                                  e_A3[p];
                 FROM   points AS p1
                 WHERE  p1.cloud = $1.cloud
                 AND    p1.label <> $1.label
                 AND    NOT EXISTS(
                          SELECT 1
                          FROM   points AS p2
                          WHERE  left_of(p2.x, p2.y,
                                         $1.x, $1.y,
                                         p1.x, p1.y)
                          AND    p2.cloud = $1.cloud)
6          IF $1.label = $2.label                          e_C1[p0,p]
7          THEN STOP
8          ELSE JUMP repeat
9  }
```

Figure 2: Transcription of Figure 1 into an imperative program. Boxes e[$v_1, v_2, \ldots$] hold parameterized SQL: v_i replaces $\$i$ in the box (otherwise, boxes are opaque to Flummi).

such programs in terms of an internal graph-based representation and is not bound to any specific frontend language syntax.

Flummi compiles into plain SQL. We describe a compiler that translates imperative programs into plain SQL queries. Looping control flow is mapped into recursive *common table expressions* (CTEs) which developers typically find hard to formulate manually [3, 9]. The compilation strategy has been deliberately designed to exploit the capabilities of modern vectorizing, multi-threaded database engines: the emitted SQL query for program `giftwrap` can evaluate the imperative code for multiple `cloud` arguments in parallel, but allows each program invocation to follow its individual control flow path (*batching* [20]). To exemplify, assume that table `clouds` holds n cloud identifiers such that the SQL query

```
SELECT c AS cloud, p AS hull
FROM   clouds AS c, LATERAL giftwrap(c) AS p
```

leads to n invocations of `giftwrap`. SQL backends that implement contemporary *query decorrelation* strategies (as found in, *e.g.*, *Umbra* or *DuckDB* [32, 33]) will be able

(1) to evaluate the Flummi-generated SQL code for `giftwrap` *for all n arguments in a single batch* (automatically forming the proper (c, p) result pairs), and
(2) save repeated evaluation effort should table `clouds` contain duplicate cloud identifiers c.

The potential of the underlying parallel database engine comes to bear on the execution of iterative imperative code, thus performing the *computation as close to the data* as possible.

We follow up on our earlier work on PL/SQL-to-SQL compilation [22, 23] and entirely redesign the compilation strategy to embrace the parallel plan evaluation capabilities found in contemporary database engines. In deviation from [23], Flummi's approach

- uses fewer compilation phases as well as intermediate code representations and thus is easier to describe and implement (see Section 2),
- holds up in the face of very complex control flow (*e.g.*, as found in the imperative code for a complete ray tracer),
- supports table-valued programs that can stream their result rows, avoiding the materialization of sizable intermediate results,
- uses a code generation strategy that facilitates the efficient execution of sequential as well as looping control flow, and
- emits SQL code that performs well on parallel query engines, in particular if they implement query decorrelation and plan evaluation in row batches (Section 3). In consequence, Flummi-generated SQL code may outperform UDF language implementations native to those engines.

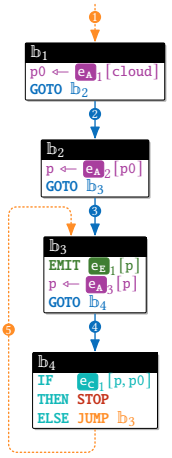


Figure 3: CFG for program `giftwrap` of Figure 2. Nodes $\mathbb{b}_i$ mark the basic blocks. Execution starts in block $\mathbb{b}_1$. Block sequencing is made explicit in terms of `GOTO`s. Block $\mathbb{b}_4$ implements the `JUMP` back to label `repeat` in Line 4 of Figure 2.

2 COMPILING PROGRAMS INTO PLAIN SQL

Flummi itself is agnostic to the specifics of user-facing program syntax. Instead, the compiler consumes programs in terms of their *control flow graph* (CFG). The CFG for the `giftwrap` code of Figure 2 is reproduced in Figure 3. Lowering imperative programs into CFGs has been well-charted territory [1] for decades. We do not explore it here.

CFG in, read-only SQL out. In Figure 2, we have deliberately chosen a simplistic variant of an imperative language whose syntactic elements are directly reflected in a CFG. However, any frontend language will fit the bill as long as it

- relies on the staples of the imperative programming paradigm (destructive variable assignment, statement sequencing, means to express arbitrary branching and iterative control flow, and result emission), and
- allows to invoke SQL queries to access database-resident data and perform computation (cf. the black boxes ▢ above).

Since Flummi will embed the contents of these black boxes into a *single read-only* SQL query, ▢ may not contain side-effecting constructs (such as transaction control, SQL DML statements, or the execution of dynamic SQL strings). Programs that explicitly raise and catch exceptions have CFG-based equivalents that Flummi can compile. Implicit exceptions (*e.g.*, division by zero), however, can not be caught by the generated SQL query and thus lead to a runtime error.

Tying the knot using a recursive CTE. Given a program's CFG, the compiler derives plain SQL code in two stages (see Figure 4):

① Assignment statements are added to the CFG's basic blocks to make data flow between blocks explicit.
② Fragments of SQL code can then be derived directly from the augmented CFG: each basic block $\mathbb{b}_i$ is mapped into its associated SQL CTE, coined $\mathbb{q}(\mathbb{b}_i)$ below.

To generate SQL code for an arbitrary CFG G, we assemble the CTEs $\mathbb{q}(\mathbb{b}_i)$ of its basic blocks to form a single SQL query $\mathbb{q}(G)$. Figure 5 sketches the general structure of $\mathbb{q}(G)$. Here, to make ideas

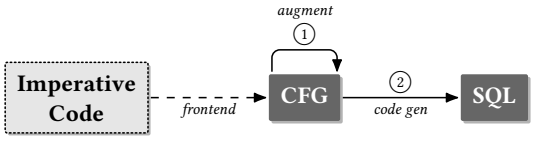


Figure 4: Compiler stages ◯ and program representations.

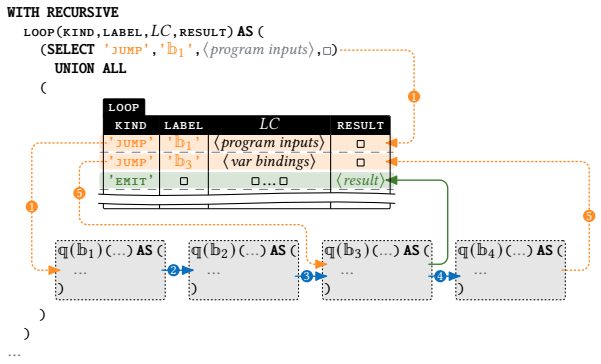


Figure 5: Structure of the SQL code generated for the CFG in Figure 10. Columns *LC* hold the bindings of the input program's loop-carried variables. Edges denote inter-CTE data flow. (Annotations in ● relate to Figure 3.)

tangible, we have instantiated $\mathtt{q}(G)$ to match the CFG for `giftwrap` of Figure 3. The construction of $\mathtt{q}(G)$ follows these principles:

- We implement *straight-line or branching control flow* in terms of inter-CTE references (blue edges ●→ in the CFG and in Figure 5): if block $\mathtt{b}_i$ precedes $\mathtt{b}_j$, in the CFG, CTE $\mathtt{q}(\mathtt{b}_j)$ refers to $\mathtt{q}(\mathtt{b}_i)$.
- Under this scheme, loops in the CFG (edges ●→) would lead to cyclic CTE references. To avoid these, we encode *looping control flow* using a *recursive CTE*[1] with *working table* `LOOP`. A row ('`JUMP`', '$\mathtt{b}_i$', ⟨*var bindings*⟩, □) in this table indicates that control has to jump to block $\mathtt{b}_i$. (In that table and in what follows, symbol □ abbreviates `NULL`.) We arrange $\mathtt{q}(G)$ such that all $\mathtt{q}(\mathtt{b}_i)$ can refer to table `LOOP`.
- In the case of `giftwrap`, CTE $\mathtt{q}(\mathtt{b}_3)$ will check table `LOOP` for rows ('`JUMP`', '$\mathtt{b}_3$', ⟨*var bindings*⟩, □)) and evaluate under all variable bindings provided by these rows. CTE $\mathtt{q}(\mathtt{b}_4)$ will return a row ('`JUMP`', '$\mathtt{b}_3$', ⟨*var bindings*⟩, □) to direct control to $\mathtt{b}_3$ in the next iteration of the recursive CTE. Together, this implements the jump from $\mathtt{b}_4$ and $\mathtt{b}_3$ along edge ❺→.

$$
\begin{array}{lll}
B ::= \texttt{EMIT}\ \widetilde{E}\ \ \widetilde{v \leftarrow E}\ \ C & & \text{basic block} \\
C ::= \texttt{STOP} & & \text{program termination} \\
\quad \mid \texttt{GOTO}\ \mathtt{s} & & \text{straight-line/branching control flow} \\
\quad \mid \texttt{JUMP}\ \mathtt{s} & & \text{looping control flow} \\
\quad \mid \texttt{IF}\ E\ \texttt{THEN}\ C\ \texttt{ELSE}\ C & & \text{conditional} \\
E ::= \mathit{SQL}\,[v, \ldots, v] & & \text{parameterized scalar SQL expressions}
\end{array}
$$

Figure 6: Statements B describe a computation step inside a basic block. Label $\mathtt{s}$ identifies the successor block, v represents variable names. (We use $\widetilde{x}$ to indicate repetitions of x.)

[1] In a nutshell, the recursive CTE (introduced with SQL:1999 [13, 41])

```
WITH RECURSIVE t(···) AS (q1 UNION ALL q∞(t))
TABLE t; -- returns the union table
```

iterates the evaluation of query q_∞ (see the pseudo code on the right). q_∞ may refer to *working table* t to access the rows produced in the immediately preceding iteration (the first iteration processes the rows produced by query q_1). Iteration stops once q_∞ returns no rows. The overall result (or: *union table*, table u on the right) then holds all rows produced in any of the iterations.

```
1 t ← q1
2 u ← t
3 repeat
4 |  t ← q∞(t)
5 |  u ← u ⊎ t
7 until t = ∅
8 return u
```

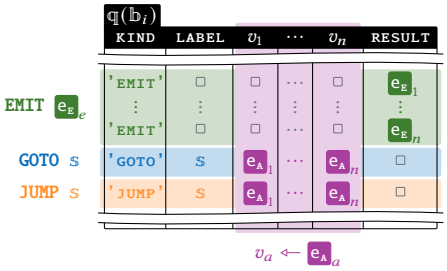


Figure 7: Tabular encoding of the output of a basic block $\mathtt{b}_i$.

In transitioning from CFG G to its SQL query $\mathtt{q}(G)$, we thus have **turned control flow into data flow**, the former based on `GOTO` and `JUMP`, the latter based on inter-CTE references: note how control and data flow edges ❺→ with identical annotations x in Figures 3 and 5 relate. (Flummi derives its name from working table `LOOP` and its instrumental role in encoding *jumps* in control flow.)

2.1 Basic Block in a CFG ≡ CTE in SQL

A basic block $\mathtt{b}_i$ in a CFG represents one small step of the computation performed by the overall imperative program. Such a step comprises, strictly in order,

1. the emission of scalar output values $\mathtt{e}_{\mathtt{E}_e}$ (via `EMIT`),
2. the evaluation of *all* embedded SQL expressions $\mathtt{e}_{\mathtt{A}_a}$ *before* the resulting values are assigned to variables v_a (via ⇦), and finally
3. the (conditional) control flow transition via `GOTO` or `JUMP` to the successor block of $\mathtt{b}_i$, say block $\mathtt{s}$.

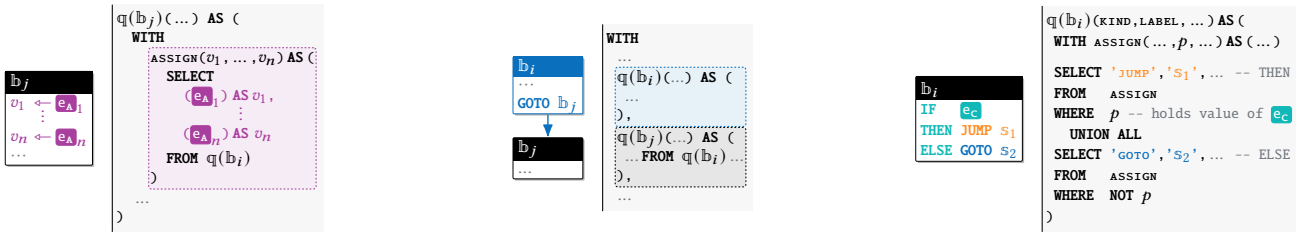
The grammar of Figure 6 restricts the acceptable statements inside a basic block to ensure that each block indeed describes a computation step of the above form.

After compilation to SQL, the CTE $\mathtt{q}(\mathtt{b}_i)$ for basic block $\mathtt{b}_i$ encodes the result of a computation step in a table as depicted in Figure 7. Query $\mathtt{q}(\mathtt{b}_i)$ yields rows with

- `KIND` '`EMIT`' for each `RESULT` value $\mathtt{e}_{\mathtt{E}_e}$ output by $\mathtt{b}_i$ (a basic block may output more than one value), and
- `KIND` '`GOTO`' or '`JUMP`' for a control flow transition to the successor block with `LABEL` $\mathtt{s}$. In block $\mathtt{s}$, variable v_a will hold the assigned scalar value $\mathtt{e}_{\mathtt{A}_a}$ as indicated by v_a's like-named column.

(No) data flow within a basic block. The immediate correspondence of basic blocks $\mathtt{b}_j$ and their CTEs $\mathtt{q}(\mathtt{b}_j)$ simplifies Flummi's compilation strategy (we reap the benefits when we address SQL code generation in Section 2.3) but also has an effect on the variable-based data flow. To see this, consider Figure 8a which shows the fragment of $\mathtt{q}(\mathtt{b}_j)$ that computes new bindings for the program variables $v_1, \ldots, v_n$: following the SQL semantics, the **SELECT** clause evaluates all embedded expressions $\mathtt{e}_{\mathtt{A}_a}$ *in parallel and/or in arbitrary order*. When the $\mathtt{e}_{\mathtt{A}_a}$ refer to variables, they thus see the bindings established by $\mathtt{b}_j$'s predecessor block $\mathtt{q}(\mathtt{b}_i)$ but *do not* observe the assignments performed in block $\mathtt{b}_j$ itself (these will only be visible in $\mathtt{b}_j$'s successor): there is no data flow *within* a basic block and we may consider its assignment and `EMIT` statements to be independent of each other.

CTEs chains express non-looping control flow. As long as basic blocks $\mathtt{b}_i$ and $\mathtt{b}_j$ connect via straight-line or branching control



(a) Evaluation of variable assignments in CTE $\mathtt{q}(\mathbb{b}_j)$.

(b) CTE-based basic block chaining (no loops).

(c) `UNION ALL` implements the `THEN`/`ELSE` branches of a conditional.

Figure 8: Flummi translates (features of) CFG blocks into SQL CTEs.

flow (edges ⟶ originating in `GOTO`s in Figure 3), we chain their associated CTEs $\mathtt{q}(\mathbb{b}_i)$ and $\mathtt{q}(\mathbb{b}_j)$ to form one executable SQL query (see Figure 8b). Only in the presence of loops, SQL code generation will resort to a *recursive* CTE to realize the iterative computation expressed by the, then cyclic, CFG.

Conditional branching control flow. A basic block $\mathbb{b}_i$ outputs rows with `KIND 'GOTO'` or `'JUMP'` to direct control flow to (one of) its successors, cf. Figure 7. To implement conditional control flow `IF` e_c `THEN...ELSE...`, we bind the Boolean value of embedded expression e_c to program variable p. SQL code generation then employs a **`UNION ALL`** set operation as shown in Figure 8c. The legs of the union are guarded by mutually exclusive predicates p (realizes the **`THEN`** branch) and **`NOT`** p (**`ELSE`**): exactly one leg will contribute to $\mathbb{b}_i$'s output for each binding of variable p.

2.2 Stage ①: Making Data Flow Explicit

Consider basic block $\mathbb{b}_4$ in Figure 3. The **`SELECT`** clause of its associated CTE $\mathtt{q}(\mathbb{b}_4)$ will evaluate embedded expression e_{c_1} to determine how to branch. Since `p` and `p0` are parameters (or: free variables) in e_{c_1}, $\mathbb{b}_4$ relies on its predecessor block $\mathbb{b}_3$ (and thus its query $\mathtt{q}(\mathbb{b}_3)$) to provide both variables as input. In essence, $\mathtt{q}(\mathbb{b}_4)$ will read:

```
SELECT ... , (e_c1[p,p0]), ...
FROM   q(b3) AS INPUTS(...,p,p0,...)
```

blockInputs(G) :
$J \leftarrow \{\mathbb{b}_1\} \cup \{\text{blocks in } G \text{ targeted by } \texttt{JUMP}\text{s}\}$
for each $\mathbb{b} \in G$ [a]
  inputs$[\mathbb{b}] \leftarrow \mathrm{FV}(\mathbb{b})$
repeat
  for each $\mathbb{b} \in G$ [b]
    inputs′$[\mathbb{b}] \leftarrow$ inputs$[\mathbb{b}] \cup (\bigcup_{\mathbb{s} \in \mathrm{succ}(\mathbb{b})}$ inputs$[\mathbb{s}] \setminus \mathrm{BV}(\mathbb{b}))$
  inputs $\leftarrow$ inputs′
  $LC \leftarrow \bigcup_{\mathbb{j} \in J}$ inputs$[\mathbb{j}]$ [c]
  for each $\mathbb{j} \in J$
    inputs$[\mathbb{j}] \leftarrow LC$
until inputs$[\cdot]$ unchanged
output $\langle$inputs, $LC\rangle$

Figure 9: Algorithm *blockInputs*(G) derives the set of input variables inputs$[\mathbb{b}]$ **for all basic blocks** $\mathbb{b}$ **in CFG** G.

Block $\mathbb{b}_3$ indeed binds `p` and thus can provide it as input to $\mathbb{b}_4$. To also provide `p0`, however, $\mathbb{b}_3$ has to receive the variable as input from its predecessors $\mathbb{b}_2$ and $\mathbb{b}_4$ in turn.

In general, the required *inputs* of a block $\mathbb{b}$ comprise both
(a) the set of its own free variables $\mathrm{FV}(\mathbb{b})$, plus
(b) the inputs of all of $\mathbb{b}$'s successors (minus the set $\mathrm{BV}(\mathbb{b})$ of variables bound and thus provided by $\mathbb{b}$ itself).

Algorithm *blockInputs*(G) in Figure 9 iterates this derivation of block inputs until it arrives at a stable mapping inputs$[\mathbb{b}]$ that provides the set of input variables for all blocks $\mathbb{b}$ in the given CFG G. Following step a above, the algorithm uses the free variables $\mathrm{FV}(\mathbb{b})$ to seed inputs$[\mathbb{b}]$ and then iterates step b until inputs$[\cdot]$ reaches a fixpoint. We have tagged the pseudo code regions corresponding to both steps with a and b in Figure 9.

From the beginning of Section 2 recall that we will bank on a recursive CTE to implement the looping control flow expressed by `JUMP` edges. Indeed, the generated SQL code will use a *single* recursive CTE to implement *all* loops in the CFG (Section 2.3 unfolds the details). Algorithm *blockInputs* prepares this code generation step in its region c in which the inputs of all basic blocks targeted by a `JUMP` (these blocks plus the program entry block $\mathbb{b}_1$ are collected in set J) are aligned to be the set of loop-carried variables LC. A single CTE working table `LOOP` with schema `LOOP(KIND|LABEL|`LC`|RESULT)` (recall Figures 5 and 7) will thus suffice to encode the inputs flowing into any `JUMP` target block.

A trace of *blockInputs* applied to `giftwrap`'s CFG in Figure 3 shows that inputs$[\cdot]$ reaches a fixpoint after two iterations (see Table 1). For reference, the top of the trace lists the sets of free/bound variables $\mathrm{FV}(\cdot)/\mathrm{BV}(\cdot)$ in each block, $\mathrm{succ}(\cdot)$ represents the CFG's control flow edges. The trace's bottom line shows inputs$[\mathbb{b}_4]$ = {`cloud`, `p`, `p0`}: besides `p` and `p0` as discussed above, note that this set additionally contains `cloud`, an input required by all `JUMP` targets in set J, including $\mathbb{b}_4$'s successor $\mathbb{b}_3$.

To complete compilation stage ①, we amend each basic block $\mathbb{b}$ in the CFG by variable assignments that render $\mathbb{b}$'s obligation to provide the inputs required by all its successors explicit. Let us denote block $\mathbb{b}$'s obligations by

$$\text{outputs}[\mathbb{b}] \coloneqq \bigcup_{\mathbb{s} \in \mathrm{succ}(\mathbb{b})} \text{inputs}[\mathbb{s}]$$

(see the bottom of Table 1 where we have listed outputs$[\cdot]$ for reference). Any variable $v \in \text{outputs}[\mathbb{b}] \setminus \mathrm{BV}(\mathbb{b})$ that is not already bound in $\mathbb{b}$ is explicitly re-assigned via $v \leftarrow \$1[v]$. This does

Table 1: A trace of inputs[·] when *blockInputs* is applied to the CFG of Figure 3. Here, the JUMP targets set is $J = \{\mathbb{b}_1, \mathbb{b}_3\}$ (both blocks are marked by superscript ^{J} below).

		$\mathbb{b}_1^J$	$\mathbb{b}_2$	$\mathbb{b}_3^J$	$\mathbb{b}_4$
FV(·)		{cloud}	{p0}	{p}	{p, p0}
BV(·)		{p0}	{p}	{p}	∅
succ(·)		{$\mathbb{b}_2$}	{$\mathbb{b}_3$}	{$\mathbb{b}_4$}	{$\mathbb{b}_3$}
Iteration					
0	a inputs[·]	{cloud}	{p0}	{p}	{p, p0}
1	b	{cloud}	{p0}	{p, p0}	{p, p0}
	c	{cloud, p, p0}	{p0}	{cloud, p, p0}	{p, p0}
2	b	{cloud, p, p0}	{cloud, p0}	{cloud, p, p0}	{cloud, p, p0}
	c	{cloud, p, p0}	{cloud, p0}	{cloud, p, p0}	{cloud, p, p0}
inputs[·]		{cloud, p, p0}	{cloud, p0}	{cloud, p, p0}	{cloud, p, p0}
outputs[·]		{cloud, p0}	{cloud, p, p0}	{cloud, p, p0}	{cloud, p, p0}

not alter the meaning of the program. If we amend the CFG for `giftwrap` (Figure 3), we arrive at the CFG of Figure 10.

With the variable-based data flow between blocks now made explicit, the CFG is ready for SQL code generation in stage ②. We turn to this next.

2.3 Stage ②: Generating SQL Code for a CFG

Below, we shed light on how

(1) a **block-level mapping** (↦) translates each basic block $\mathbb{b}_i$ in G into its associated CTE $\mathfrak{q}(\mathbb{b}_i)$, before

(2) a **program-level mapping** (⇛) assembles these pieces into the encompassing query $\mathfrak{q}(G)$ that implements the imperative program as a whole.

2.3.1 Block-level SQL code generation (mapping ↦).

We define mapping ↦ in terms of inference rule Block of Figure 11a. Input to this rule is a single prototypical basic block $\mathbb{b}$, featuring output emission (via EMIT), variable assignments (⇦), outgoing control flow (represented by c), as well as predecessor blocks that either GOTO (the blocks $\mathbb{g}_g$) or JUMP to $\mathbb{b}$.[2]

We build query $\mathfrak{q}(\mathbb{b})$ such that it realizes the computation step performed by basic block $\mathbb{b}$ (recall Section 2.1). On execution, $\mathfrak{q}(\mathbb{b})$ returns a table of the shape shown in Figure 7. Below we use 0 ... 3 to refer to the relevant parts of Rule Block.

0 + 0. Collect inputs from predecessor blocks. Query $\mathfrak{q}(\mathbb{b})$ retrieves the bindings for the input variables in set $I ::= \text{inputs}[\mathbb{b}]$ from the tables computed by $\mathbb{b}$'s predecessor blocks. These predecessors comprise (1) the blocks $\mathbb{g}_g$ that GOTO-transition to $\mathbb{b}$ (their outputs are available in tables $\mathfrak{q}(\mathbb{g}_g)$) as well as (2) the blocks that JUMP to $\mathbb{b}$ (their output is collectively found in working table LOOP of the recursive CTE). Predicates LABEL='$\mathbb{b}$' ensure that $\mathfrak{q}(\mathbb{b})$ only grabs those inputs directed to block $\mathbb{b}$. These inputs are collected in local CTE INPUTS ready to be read in 1 and 2 below.

[2] To compactly render these mappings, we adopt Steele's notational conventions [42]. Sets and their elements are denoted by uppercase and their associated lowercase letters, respectively: $X = \{x_1, x_2, x_3, \ldots\}$. When we enumerate the elements x_i, we index by i and specify separator ⨟ (if omitted, the separator defaults to comma ','): $\widetilde{x_i}^{\,i=n..m}_{⨟} = x_n ⨟ x_{n+1} ⨟ \cdots ⨟ x_m$. If the range for the index is omitted, the containing set defines the enumeration: if $X = \{x_1, x_2, x_3\}$, then $\widetilde{x_j}^{\,j} = x_1, x_2, x_3$.

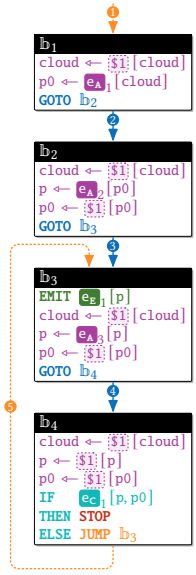


Figure 10: CFG for program `giftwrap` (derived from Figure 3) after variable-based data flow has been made explicit: variable assignments encode all outputs that a basic block is required to provide to its successors. This CFG is input to SQL code generation. (Annotations in ● relate to Figure 5.)

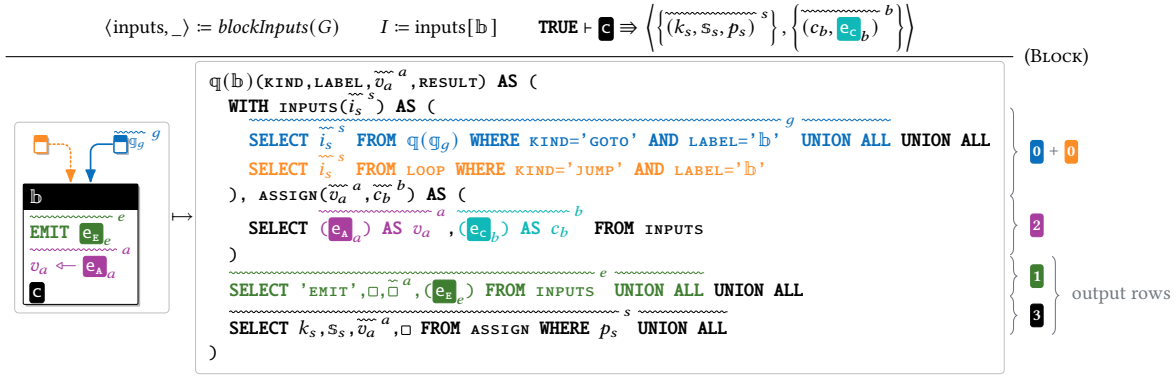
1. Emit scalar output values. Based on CTE INPUTS, query $\mathfrak{q}(\mathbb{b})$ evaluates all scalar expressions e_{E_e} to be emitted by block $\mathbb{b}$. Each resulting value is placed in the RESULT column of an output row tagged with KIND 'EMIT' (all other columns are set to □, cf. Figure 7).

2. Evaluate expressions, assign to variables. The rows of CTE INPUTS are read to compute the values of all embedded expressions e_{A_a} and e_{C_b} found in $\mathbb{b}$. The former are assigned to the variables v_a (recall Figure 8a in Section 2.1), the latter are the conditions in c that guide control flow. Since the CFG has been amended with explicit variable assignments (Section 2.2), the resulting local CTE ASSIGN is guaranteed to hold all inputs required by $\mathbb{b}$'s successor blocks.

3. Conditional control flow transition. Finally, $\mathfrak{q}(\mathbb{b})$ outputs rows that direct control flow to the successor blocks of $\mathbb{b}$. To this end, the new variable bindings collected in CTE ASSIGN are placed in rows tagged with KIND $k_s \in$ {'GOTO', 'JUMP'} targetting the successor with LABEL $\mathbb{s}_s$ under condition p_s. To determine the triples $(k_s, \mathbb{s}_s, p_s)$, Rule Block invokes the auxiliary mapping ⇛ (defined in Figure 11b) on the final statement c in $\mathbb{b}$. Statement c adheres to the non-terminal C of the grammar in Figure 6 and may thus contain IF-THEN-ELSE conditionals nested to arbitrary depth. To illustrate, if c is the IF-THEN-ELSE statement of block $\mathbb{b}_4$ of the CFG in Figure 10, we have

$$\texttt{TRUE} \vdash \texttt{c} \Rrightarrow \left\langle \{(\overbrace{\texttt{'JUMP'}}^{k_1}, \overbrace{\texttt{'}\mathbb{b}_3\texttt{'}}^{\mathbb{s}_1}, \overbrace{\texttt{TRUE AND NOT } c_1}^{p_1})\}, \{(c_1, e_{C_1})\} \right\rangle$$

Mapping ⇛ returns the triple $(k_1, \mathbb{s}_1, p_1)$ and introduces column c_1 to hold the Boolean value of condition e_{C_1} (e_{C_1} is placed in c_1 by CTE ASSIGN, see 2 above). With predicate p_1 = TRUE AND NOT c_1,



(a) Definition of block-level mapping $\mathbb{b} \mapsto \mathbb{q}(\mathbb{b})$ (Rule Block).

$$\frac{}{p \vdash \texttt{GOTO}\ \mathbb{s} \Rrightarrow \langle\{(\texttt{'GOTO'}, \texttt{'}\mathbb{s}\texttt{'}, p)\}, \varnothing\rangle} \qquad \frac{}{p \vdash \texttt{JUMP}\ \mathbb{s} \Rrightarrow \langle\{(\texttt{'JUMP'}, \texttt{'}\mathbb{s}\texttt{'}, p)\}, \varnothing\rangle}$$

$$\frac{}{p \vdash \texttt{STOP} \Rrightarrow \langle\varnothing, \varnothing\rangle}$$

$$\frac{\begin{array}{c} c \coloneqq \text{fresh column name} \\ (p\ \texttt{AND}\ \ \ c) \vdash C_1 \Rrightarrow \langle s_1, b_1\rangle \\ (p\ \texttt{AND NOT}\ c) \vdash C_2 \Rrightarrow \langle s_2, b_2\rangle \\ s \coloneqq s_1 \cup s_2 \qquad b \coloneqq b_1 \cup b_2 \cup \{(c, e_c)\} \end{array}}{p \vdash \texttt{IF}\ e_c\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2 \Rrightarrow \langle s, b\rangle}$$

(b) Auxiliary mapping $p \vdash c \Rrightarrow \langle\cdot,\cdot\rangle$ (invoked by $\mapsto$) compiles control flow statement c under condition p.

Figure 11: Flummi's stage ②: compiling prototypical basic block $\mathbb{b}$ to SQL code fragment $\mathbb{q}(\mathbb{b})$.

query $\mathbb{q}(\mathbb{b}_4)$ will output a row with (KIND, LABEL) = ('JUMP', '$\mathbb{b}_3$') only if c_1 holds FALSE, thus meeting the expected behavior of the IF-THEN-ELSE in block $\mathbb{b}_4$. Note how the THEN STOP branch of the conditional does *not* contribute such a triple: the next iteration of CTE LOOP will thus not perform a GOTO/JUMP on STOP's behalf and control flow will end for all TRUE values in column c_1.

All rows produced in 1 and 3 are combined via UNION ALL to form the overall output of $\mathbb{q}(\mathbb{b})$. For reference, Figure 12 shows $\mathbb{q}(\mathbb{b}_3)$, the result of applying the block-level mapping $\mapsto$ to basic block $\mathbb{b}_3$ of the CFG in Figure 10.

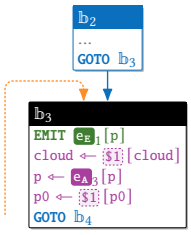
2.3.2 Program-level SQL code generation (mapping $\Rrightarrow$).

Inference Rule Program of Figure 13 defines mapping $\Rrightarrow$ which generates SQL code for the CFG G of a complete program. This mapping first

(1) invokes block-level code generation for all basic blocks $\mathbb{b}_1, \ldots, \mathbb{b}_n$ in G (Rule Program is *compositional*: the n invocations of $\mapsto$ are independent of each other and may happen in any order or even in parallel), then
(2) wraps the resulting CTEs $\mathbb{q}(\mathbb{b}_1), \ldots, \mathbb{q}(\mathbb{b}_n)$ in recursive CTE LOOP to establish the required infrastructure for the evaluation of looping control flow.

At query runtime, CTE LOOP iteratively evaluates the queries $\mathbb{q}(\mathbb{b}_1), \ldots, \mathbb{q}(\mathbb{b}_n)$. After each such iteration, we collect the rows of KIND 'JUMP' and 'EMIT' output by the blocks in sets L and E, respectively (for the giftwrap CFG of Figure 10, the "jumpers" and "emitters" are $L = \{\mathbb{b}_4\}$ and $E = \{\mathbb{b}_3\}$, respectively). The semantics of recursive CTEs ensure that

- upon the CTE's next iteration, all successors $\mathbb{s}$ of the blocks in L find rows with (KIND, LABEL) = ('JUMP', '$\mathbb{s}$') in *working table* LOOP, and that




```
q(𝕓3)(KIND,LABEL,cloud,p,p0,RESULT) AS (
  WITH INPUTS(cloud,p,p0) AS (
    SELECT cloud,p,p0
    FROM   q(𝕓2)
    WHERE  KIND='GOTO' AND LABEL='𝕓3'
      UNION ALL
    SELECT cloud,p,p0
    FROM   LOOP
    WHERE  KIND='JUMP' AND LABEL='𝕓3'
  ), ASSIGN(cloud,p,p0) AS (
    SELECT (cloud) AS cloud,
           (e_A_3) AS p,
           (p0) AS p0
    FROM   INPUTS
  )
  SELECT 'EMIT',□,□,□,□,□,(e_E_1)
  FROM   INPUTS
    UNION ALL
  SELECT 'GOTO','𝕓4',cloud,p,p0,□
  FROM   ASSIGN
  WHERE  TRUE -- unconditional GOTO
)
```

Figure 12: Block-level mapping $\mapsto$ (defined in Figure 11) generates SQL code fragment $\mathbb{q}(\mathbb{b}_3)$ for basic block $\mathbb{b}_3$.

- the union of all rows output during the computation is available once CTE LOOP completes. The rows with KIND 'EMIT' in the *union table* collectively represent the result of the program run.

2.4 Batched Program Execution

It is a staple of the SQL code generated by Rule Program that it realizes execution for an *entire batch of* $s \geqslant 1$ *program inputs*. Each row returned by the CTEs $\mathbb{q}(\mathbb{b}_i)$ and LOOP tracks the program state—current CFG block and variable bindings in columns {LABEL} ∪ LC—of one of these executions. Each such execution may independently follow its own control flow path, return results, or terminate.

Let us evaluate the generated SQL code $\mathbb{q}$(giftwrap(c)) for program giftwrap over an input batch of arguments cloud ∈ $\{A, B, C\}$ (with batch size $s = 3$) as follows:

$$\frac{\langle \text{inputs}, LC\rangle \coloneqq \mathit{blockInputs}(G) \qquad \overbrace{\square_{\mathbb{b}_b} \mapsto \texttt{CTE}_b}^{b=1..n} \qquad L \coloneqq \{\text{blocks in } G \text{ performing } \texttt{JUMP}\text{s}\} \qquad E \coloneqq \{\text{blocks in } G \text{ performing } \texttt{EMIT}\text{s}\}}{\square \Longmapsto \texttt{WITH RECURSIVE LOOP(KIND,LABEL,}\overbrace{lc_i}^{i}\texttt{,RESULT) AS (}\ldots\texttt{)}\ \texttt{SELECT RESULT FROM LOOP WHERE KIND='EMIT'}} \ (\textsc{Program})$$

Figure 13: Flummi's stage ②: progam-level mapping $G \Longmapsto \mathfrak{q}(G)$: assembling block-level SQL fragments using recursive CTE LOOP.

```
SELECT  c AS cloud, p AS hull
FROM    (VALUES (A),(B),(C)) AS c,
LATERAL q(giftwrap(c)) AS p
```

We obtain the trace of intermediate CTE results reproduced in Figure 14. (Here we assume that table `points` holds point clouds with identifiers A, B, and C whose convex hulls contain points $\{a_1, a_2, a_3, a_4, \ldots\}$, $\{b_1\}$, and $\{c_1, c_2\}$, respectively.) Program execution starts in basic block $\mathbb{b}_1$ for a batch of size $s = 3$ rows with (KIND, LABEL) = ('JUMP', '$\mathbb{b}_1$'). Through the iterations ⓘ--- ($i = 1, 2, \ldots$) of the recursive CTE LOOP, the trace shows the results returned by the CTEs $\mathfrak{q}(\mathbb{b}_i)$. The subset of rows marked by ◇ form the contents of working table LOOP: these rows define the program output (KIND = 'EMIT', see column RESULT) or direct control flow back to loop entry block $\mathbb{b}_3$ (KIND = 'JUMP').

We note the following:

- If we consider the bindings for row variable c (see the grey rightmost column in Figure 14), the 'EMIT' rows indeed define the expected result set of (c, p) pairs $\{(A, a_1), (A, a_2), \ldots, (B, b_1), (C, c_1), (C, c_2)\}$.
- Program executions for a batch of arguments are independent: execution for argument cloud = B stops before execution for argument C stops while execution for argument A continues.
- $\mathfrak{q}(\mathbb{b}_1)$ and $\mathfrak{q}(\mathbb{b}_2)$ merely return the empty row set ∅ once the control flows for all executions have entered the loop $\mathbb{b}_3$–$\mathbb{b}_4$ in iterations 2, 3, ... of the recursive LOOP CTE.
- Only the JUMPs back to loop entry block $\mathbb{b}_3$ end up in working table LOOP (GOTO-based block transitions do not).

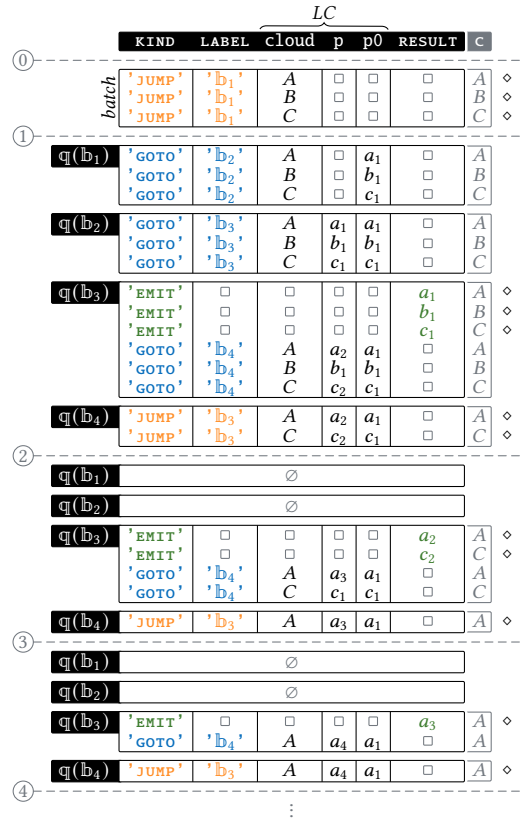
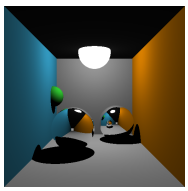


Figure 14: Intermediate CTE results of the SQL code for `giftwrap` when invoked on batch $\{A, B, C\}$ of `cloud` arguments (ⓘ--- indicates the i-th iteration of recursive CTE LOOP).

3 EXPERIMENTS WITH 17 PROGRAMS

Our focus has exclusively been on program `giftwrap` until now, but this section will study the runtime behavior of a collection of 17 imperative programs (see Table 2) of varying complexity, from a loop-less few-liner to a complete ray tracer. The ray tracer `ray` renders the 512×512 pixels of the image on the right (the scene definition has been adapted from [24]) in 5 seconds, *i.e.*, at about 20 ţs/pixel, if we draw on *Umbra* as the SQL backend [32].



An accompanying GitHub repository[3] holds the 17 imperative-style source programs, renderings of their original and optimized CFGs, as well as the Flummi-generated SQL code, ready for execution on *DuckDB*, *Umbra*, and *PostgreSQL*. We have specifically authored these programs to exercise aspects of Flummi but have also incorporated three entries from the *ProcBench* benchmark suite [19] of PL/SQL UDFs (marked with PB in Table 2). Source program size varies, with `ray` comprising about 300 lines which Flummi compiles into 3,000 lines of SQL code.

[3] https://github.com/flummi-compiler/PVLDBv17

Table 2: Collection of 17 imperative programs with code characteristics and runtimes per invocation. Measured on *DuckDB*.

	Program	CC	Loops + queries	# Blocks	# Variables total / *LC*	Runtime per Invocation [ms] 1/5 batch / 5/5 batch	Speedup	Batching sizes 1/5 … 5/5
`giftwrap`	convex hull of a point cloud	2	Q	5	3 / 3	0.162 / 0.157	(1.04×)	
`march`	track border of a 2D object	2	Q	4	5 / 4	338.704 / 76.845	(4.41×)	
`vm`	execute code on a simple VM	31	Q	67	8 / 7	24.733 / 12.290	(2.01×)	
`oil`	2D rotational line sweep	3	Q Q	8	9 / 8	86.275 / 36.429	(2.37×)	
`visible`	visibility in a hilly 3D landscape	3	Q Q	8	10 / 9	82.975 / 17.745	(4.68×)	
`force`	*n*-body simulation (Barnes-Hut tree)	4	Q Q	11	9 / 5	20.439 / 8.258	(2.48×)	
`ray`	ray tracer (adapted from [24])	5	Q Q Q Q	63	58 / 28	0.206 / 0.152	(1.36×)	
`ship`	determine preferred shipping method	1	QQQ	7	4 / 1	0.014 / 0.004	(4.04×)	
`late`	identify delayed orders (TPC-H *Q*21)	3	Q Q	9	6 / 5	938.750 / 777.450	(1.21×)	
`supply`	try to reduce supplier costs	3	Q QQ	9	7 / 4	0.010 / 0.009	(1.11×)	
`savings`	optimize supply chain of orders	3	QQ QQQQ	8	5 / 2	0.013 / 0.010	(1.25×)	
`margin`	buy/sell parts with maximum profit	6	Q Q Q	13	7 / 6	0.022 / 0.018	(1.19×)	
`sched`	schedule production of lineitems	4	QQ Q Q	13	10 / 9	0.118 / 0.041	(2.89×)	
`packing`	tightly pack lineitems into containers	5	QQ Q ␣	17	10 / 9	0.074 / 0.038	(1.98×)	
`distinct`	find unique entries in word list	2	␣	9	4 / 3	0.002 / 0.001	(1.21×)	
`profit`	sum daily net profit in date range	2	QQ	6	5 / 3	18.024 / 4.056	(4.44×)	
`promo`	conversions in promotion channels	4	QQQ	7	11 / 1	1.400 / 0.289	(4.84×)	

We compile and execute these programs to assess

- how contemporary database engines (*DuckDB* [35] and *Umbra* [32], in particular) implement **query decorrelation techniques that support the efficient batched evaluation** of Flummi-compiled programs,
- how Flummi's **CFG-centered compilation** compares against our earlier work on PL/SQL-to-SQL translation (which relied on a more involved chain of translation steps [23]),
- how the **`EMIT`-based generation of table-valued program results** fares against an array-centric programming style, and
- how Flummi-generated SQL code **admits parallel evaluation** (in contrast to *Umbra*'s native *UmbraScript* compiler [32] which translates imperative programs into single-threaded machine code).

Program characteristics. In Table 2, columns **CC** (*cyclomatic complexity*, *i.e.*, the number of independent control flow paths [28]) and **Loops** aim to measure the control flow complexity of the 17 programs. For `margin`, `Q Q ␣` indicates that the program comprises two loops nested inside each other (`␣`), with black-boxed SQL queries `Q` embedded at program top level as well as in the outer- and innermost loops. (You will find that `ship` and `promo` are the only loop-less programs.) Column **# Blocks** reports the number of basic blocks in the programs' CFGs. Since blocks equate CTEs, this count also reflects the number of $\P(\mathbb{b}_i)$ in the emitted SQL code. Destructive variable assignment is a staple of the imperative programming style. Column **# Variables** reports how many program variables are in use (the loop-carried variables in set *LC* are alive across iterations and thus are held in CTE working table `LOOP`, cf. Figures 5 and 14).

The following discussion is based on timings taken on a 64-bit Linux x86 computer with two AMD EPYC™ 7402 CPUs running at 2.8 GHz (24 cores/48 threads per CPU) and 2 TB of RAM. This machine hosted *DuckDB* v0.10.0, *Umbra* v0.1-1285, and *PostgreSQL* v16.1. The Flummi compiler itself has been implemented in Python—in the experiments, compilation times never exceeded 100 ms. For all experiments, we report the median run times of three repeated program runs.

3.1 Query Decorrelation Leads to Batching

Query decorrelation as proposed by Neumann and Kemper in [33] and implemented in, *e.g.*, *DuckDB*, *HyPer* [31], *Materialize* [29], or *Umbra*, automatically leads to the formation of batches of program arguments (Section 2.4).

The correlated occurrence of row variable `c` on the right-hand side of the **`LATERAL`** join in

```
SELECT  c AS cloud, p AS hull
FROM    clouds AS c,
LATERAL ¶(giftwrap(c)) AS p
```

suggests an iterated argument-by-argument evaluation of the invocations `giftwrap(c)`. With decorrelation, instead, the query engine follows the steps DEDUP-EVAL1-DUP:

[DEDUP] **Collect a duplicate-free table of all bindings** for argument `c` (cf. the contents $\{A, B, C\}$ of column `cloud` in the top *batch* of Figure 14).

[EVAL1] **Evaluate the plan for $\P$(`giftwrap(c)`) once** over the binding table. Decorrelation transforms the plan such that it attaches the associated binding for argument `c` to any emitted result row (recall the grey column `c` in Figure 14).

[DUP] **Replicate results** to compensate for the duplicate argument elimination in step DEDUP. Program execution over repeated arguments has thus been avoided.

The larger the batch size, the more program executions share the effort to set up and evaluate the recursive CTE in $\P$(`giftwrap(c)`). In consequence, the **Runtime per Invocation** on *DuckDB* (see Table 2) significantly drops if we grow the batch size by a factor of 5. Column **Batching** sketches how invocation time declines as batch sizes increase. (What constitutes a full batch depends on the program: for `ray`, say, we render 262,144 pixels, for `giftwrap` we compute the convex hull of 1,000 point clouds.)

Automatic program result caching. Database engines that do *not* implement decorrelation of subqueries, *e.g.*, *PostgreSQL*, will execute $\P$(`giftwrap(c)`) once per program invocation, *i.e.*, argument by argument. For programs `force`, `march`, and `packing`, Figure 15

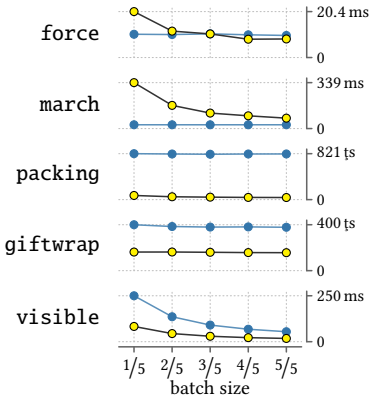


Figure 15: Time per program invocation on *DuckDB* (○-○) and *PostgreSQL* (●-●) as batches grow from 1/5 to full size.

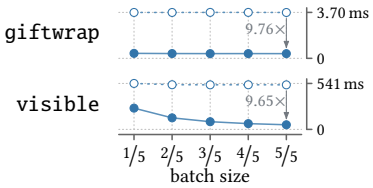


Figure 16: *PostgreSQL*'s `LATERAL` memoization improves program invocation times (○-○: memoization disabled).

indeed shows how *PostgreSQL* is unaffected by the batch size while *DuckDB* shows the expected invocation time savings. In the case of `march`, evaluation over larger batches allows *DuckDB* to catch up with *PostgreSQL*'s clever exploitation of indexes.

However, when batches contain duplicate arguments—we have specifically crafted such batches with 90 % duplicates for programs `giftwrap` and `visible`—*both backends* appear to benefit. While this is expected for *DuckDB* (and the deduplication step DEDUP built into decorrelation), for *PostgreSQL* we instead observe the effect of its memoizing implementation of **`LATERAL`** joins: results of the join's right-hand side are memoized and reused when row variable bindings reoccur on the left-hand side. For the duplicate-heavy batch, *PostgreSQL*'s `EXPLAIN ANALYZE` documents a 90 % hit rate for the system's `Memoize` plan operator. This matches the 9-fold invocation time speedup we observe for `giftwrap` and `visible` when we execute these programs with **`LATERAL`** memoization enabled (see Figure 16).

3.2 Flummi Aids Vectorizing Pipelining Engines

How does Flummi fare against our PL/SQL-to-SQL compiler developed in [23]? The latter operates on the level of individual assignment statements (rather than basic blocks) and translates sequences of n such assignments into a chain of $n-1$ **`LATERAL`** joins which, gradually and **only at query runtime**, assemble a table of bindings for the variables in scope. In consequence, the SQL code generation strategy of [23] leads to chains of dependent joins ⋈ (borrowing notation from [33]): in these joins, the evaluation of the right-hand side depends on rows delivered by the left-hand plan. Figure 17 (left) shows such a join chain for a sequence of three statements △. Modern query engines like *DuckDB* implement ⋈ using the

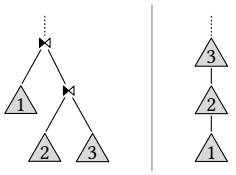


Figure 17: Plan shapes for statement sequences: [23] (left) vs. Flummi.

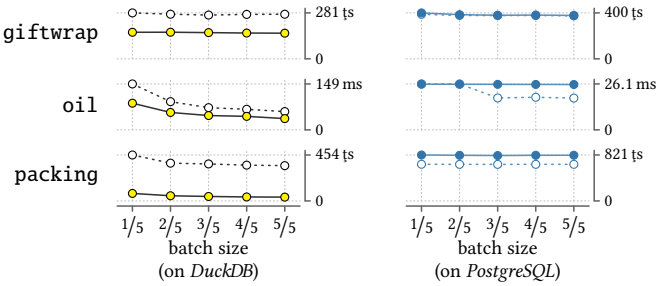


Figure 18: Invocation times observed for Flummi (●-●) and the `LATERAL`-based SQL code generator (○-○) of [23].

DEDUP-EVAL1-DUP strategy and thus materialize the results of the subplans 1△ and 2△ (in step DEDUP), disrupting pipelined plan evaluation: while deduplicating joins enable batched program evaluation as discussed in Section 3.1, they are a bane of the compilation of long statement sequences.

Flummi's SQL code generator avoids this conundrum through the linear chaining of CTEs (recall Figure 8b): sequences of `GOTO`-connected basic blocks are compiled into stacks of CTEs as shown on the right side of Figure 17. **Already at SQL code generation time**, we fix the schemata of these CTEs to hold columns for all variables in scope using Algorithm *blockInputs* of Section 2.2. Engines can inline these CTEs (*e.g.*, as *DuckDB* does) and/or stream rows through the resulting plans, facilitating vectorized and pipelined query evaluation.

These observations about plan shapes are mirrored by the program invocation times reported in Figure 18. As expected, *DuckDB* benefits from Flummi's CTE-centric compilation. *PostgreSQL* remains largely unaffected by the change of SQL code generation strategy: its query engine evaluates both the stack of CTEs emitted Flummi as well as the chain of **`LATERAL`** joins generated by [23] using a nested loops strategy.

3.3 Lump `GOTO` with `JUMP`?

Flummi translates looping (`JUMP`) and non-looping (`GOTO`) control flow differently. Clearly, we could unify both, *i.e.*, treat `GOTO` just like `JUMP`, and thus streamline SQL code generation even further. Would program run time suffer?

To this end, we patched Flummi to implement *all* control flow via `JUMP`s:

- Programs now place more rows with `KIND = JUMP` in working table `LOOP` (in the trace table of Figure 14, all `GOTO`s in column `KIND` are replaced by `JUMP`s and all rows are marked by ⋄). At **program run time**, even straight-line or branching control flow transitions now require an iteration of the recursive CTE and

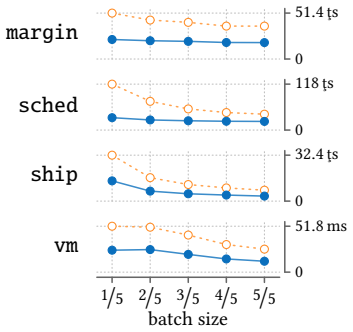


Figure 19: Implementing all control flow in terms of JUMPs (○⋯○) *vs.* Flummi's CTE chaining for straight-line/branching GOTO (○—○). Program invocation times, measured on *DuckDB*.

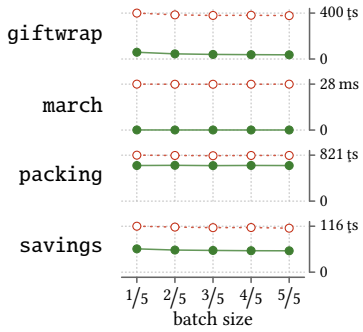


Figure 20: Streaming unblocks set-valued programs and improves response time: run time until the first EMIT (○—○) *vs.* until termination via STOP (○⋯○). Measured on *PostgreSQL*.

thus the maintenance of working and union tables. Figure 19 paints a clear picture of the resulting performance loss (measured on *DuckDB*).

- In contrast, an implementation of GOTO in terms of CTE chaining (Figure 8b) makes non-looping control flow a **query planning time** issue. The underlying SQL engine may even inline referenced CTEs into their referrer: a single query block is formed that realizes an entire sequence of program statements. Less iterations of CTE LOOP are required and program run time is reduced.

3.4 Streaming EMITs *vs.* Array-Valued Programs

Recent work on the translation of imperative code into plain SQL supports programs that return a single *scalar* result [23, 37]. To circumvent this restriction, programs resort to assemble set-valued results in bulk-typed variables—in an array `xs`, say, that is repeatedly appended to via `xs ⇐ xs + E`. Array `xs` is returned only *when the program stops*.

Result accumulators like `xs` are typically loop-carried: the—potentially sizable—array is passed from iteration to iteration by saving it into a working table. If `xs` grows element by element across n iterations, the program moves $1 + 2 + \cdots + n = 1/2 \cdot (n^2 + n)$ elements overall. This overhead can have a detrimental impact on program run time [21, 23].

In Flummi, instead, programs may repeatedly use EMIT E to return set-valued results element by element. These emitted values do not pollute the working tables of subsequent iterations (recall 0 + 0 in Figure 11a). Query engines that follow a Volcano-style or push-based execution discipline (*e.g.*, *PostgreSQL* or *DuckDB*/*Umbra*, respectively), can *immediately* pass emitted elements to the invoking query which thus is not blocked until program completion. One benefit of such a streaming implementation of set-valued programs can be observed in Figure 20: the execution time until a program issues its *first* EMIT (as opposed to: until the program STOPs) defines its response time.

3.5 Unlocking Parallelism on *Umbra*

The ongoing research work on *Umbra* [32] aims to architect a database system that supports computational workloads beyond vanilla relational queries. *Umbra*'s query compiler generates native machine code for queries formulated in both a rich dialect of SQL and the system's scripting language *UmbraScript*. Much like Flummi, *UmbraScript* encourages imperative-style programming using (1) statement sequences, (2) conditional branching, (3) updatable variables that can be bound to the results of embedded scalar SQL query blocks, and (4) cursor loops over tabular query results. The accompanying GitHub repository[3] contains *UmbraScript* formulations for a subset of the programs in Table 2.

UDFs whose bodies are expressed in terms of such scripts are invoked argument by argument, *i.e.*, no batching is performed. This severely limits the parallelism available outside of embedded SQL query blocks. Indeed, *Umbra* compiles scripts into *single-threaded* machine code and, as we write this, the system's morsel-based parallel evaluation strategy [26] thus exclusively applies to plain SQL queries. A parallelism-aware rewrite of the *UmbraScript* compiler would need to understand the interplay of batching and possibly deeply nested iteration (*e.g.*, loop patterns like ⎵⎵ or ⎵) and thus draw on *nested parallelism* techniques (*e.g.*, as explored by Blelloch [5]). Paraphrasing *Umbra*'s main developer, this "would be a major undertaking."

***Umbra* parallelizes Flummi-generated SQL.** Flummi-compiled programs, however, realize *all* computation in terms of plain SQL query blocks: *Umbra*'s parallelization strategy applies throughout, covering the evaluation of embedded queries as well as the execution of control flow. In consequence, if parallel compute resources are indeed available, we find the run time of Flummi-generated code to match or—often significantly—improve those obtained with *Umbra*'s native script compiler (see Figures 21 and 22). While Flummi can benefit from an increase in argument batch size and/or thread count on *Umbra*, it is typical for *UmbraScript* code to show a rather flat or irregular run time profile instead (*e.g.*, see programs `giftwrap` and `visible` in Figure 21). Additional threads aid *UmbraScript* only if embedded SQL queries account for a lion share of script execution time. (In Figure 21, we include program invocation times on *DuckDB* to demonstrate that Flummi's exploitation of parallelization opportunities is not tied to a specific backend.)

3.6 Head to Head: SQL Engine vs. Python

Contemporary Python has been engineered to interpret imperative programs efficiently. Paired with *DuckDB*-specific bindings, Python programs are enabled to execute embedded SQL queries within the interpreter process itself [35]. We chose this widely deployed

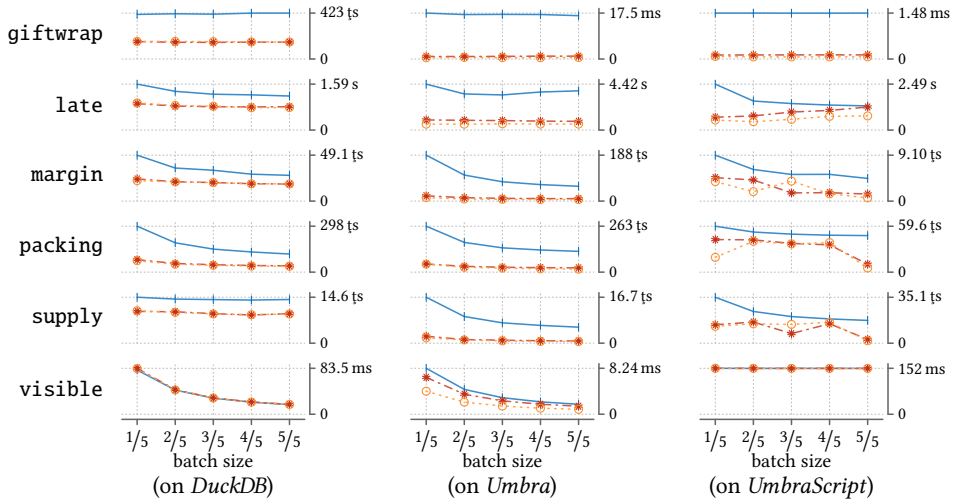


Figure 21: Flummi exploits parallelization opportunities that come with batching (1 thread, 12 threads, 24 threads). For a comparison of Flummi-generated SQL code *vs. Umbra*'s native *UmbraScript*, see Figure 22.

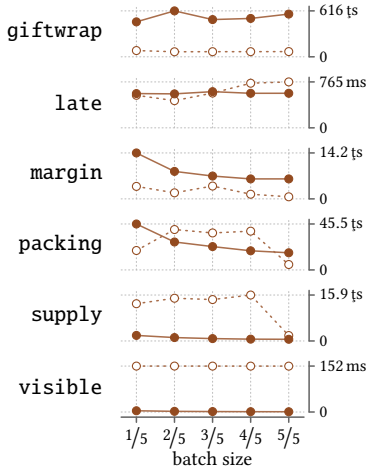


Figure 22: *Umbra*, running 24 threads: invocation times for Flummi-generated SQL ( ) *vs.* native *UmbraScript* ( ).

setup to serve as a baseline for this section's final assessment of the runtime performance of Flummi-compiled programs. To this end, we transcribed five of our sample programs of varying control flow complexity into equivalent Python functions. We kept the imperative program flavor but locally adapted the code to reflect Python's specifics (the Python code for these functions is included in the GitHub repository[3]):

- Functions rely on a function result cache to avoid redundant computation in the presence of duplicate inputs (thus mimicking the effect of Flummi's DEDUP-EVAL1-DUP through decorrelation, see Section 3.1).
- Groups of embedded SQL queries are bundled to yield a single query that returns multiple results which the function receives using Python's multi-target assignments. In `ship`, for example, we bundled all three embedded queries (effectively turning `QQQ` into `Q`, recall Table 2).

Table 3: Flummi's speedup over Python on *DuckDB*.

Program	Speedup
`giftwrap`	1.64×
`packing`	68.96×
`supply`	43.46×
`ship`	1489.57×
`distinct`	0.37×

- Function execution is parallelized using Python's builtin `multiprocessing` module which may call on up to 24 workers.

Table 3 reports that *DuckDB* evaluates the Flummi-compiled SQL code significantly faster than Python can run the equivalent functions over the same input tables. If program runtime is dominated by the embedded queries, Flummi's advantage is less pronounced (`giftwrap`). In case of a continued back and forth between Python interpretation and the evaluation of embedded SQL queries (which occurs 150,000 times across all invocations of the Python variant of `ship`), the runtime performance of Flummi's single-query SQL output is out of reach for Python. Programs that need not access database-resident data at all, however, remain better served by the runtime environment of a general-purpose programming language that has been tuned to execute imperative programs: the Python variant of program `distinct`, which iteratively performs text processing independent of any tabular data (and thus is of kind ⎵, see Table 2), executes more than twice as fast as its Flummi-compiled SQL equivalent.

We contend that SQL engines indeed can excel at the execution of imperative programs over database-resident data.

4 ADDITIONAL RELATED WORK

Flummi compiles imperative programs into plain SQL. This particular choice of *target language* has salient consequences which we share with a number of related efforts:

The universality of SQL decouples code generation from specific database engines. This is in contrast to approaches that target APIs that are backend-specific (like *UDO* [40]) or are deliberately low-level (like *Redshift*'s C++ code generator [2] or *Weld* [34]). Flummi, in particular, does not inspect or rely on the contents of the black boxes and compiles control flow into (recursive) CTEs which have been an integral part of the SQL standard for 25+ years [11, 13, 41]. Thus, further backends (*e.g.*, *Materialize* [29]) are in immediate reach for Flummi.

Programs may be executed on backends that otherwise do not offer support for user-defined computation inside the database kernel. Flummi and similar translators may bring UDF functionality to, say, *DuckDB* [35] without the need to implement a PL/SQL-style interpreter [22, 23, 37].

The decades-old wisdom of RDBMS engineering comes to bear on imperative program execution. Relational query engines make for runtimes that do not collapse when programs shuffle large data volumes, *e.g.*, machine learning workloads [4, 12, 39]. Further, the engines' set-orientation naturally supports batched and parallel program execution [20].

No back and forth between set-oriented query evaluation and statement-by-statement program execution. Frequent or costly switches between both processing modes result in disappointing UDF performance. This observation led *Froid* [36, 37] to spearhead a now very active branch of research on UDF-to-SQL compilation [7, 16, 18, 23, 43, 44]. Flummi expands on these efforts through its support for arbitrary control flow and table-valued programs (*Froid* cannot translate 13 of the 17 programs in Table 2).

SQL is sufficiently expressive to serve as the translation target for a variety of programming paradigms. This includes imperative languages like PL/SQL [12, 23, 37], Python-like scripting [14, 15], declarative languages with a focus on recursion [6, 10, 27], as well as comprehension-centric functional DSLs like *LINQ* or *Links* [8, 17, 30].

5 WRAP-UP

Imperative database programs *compute* (an aspect which we have encapsulated inside the query boxes) and *exert control flow*. We argue that CFGs provide an expressive and workable representation of both facets: Flummi's focus on CFGs led to a compilation strategy that can (1) equate the basic blocks in a CFG with non-recursive CTEs (facilitating a compact and compositional formulation of SQL code generation) and (2) build on recursive CTEs to provide a relational encoding of looping control flow that naturally copes with batching (providing a source of parallelism that contemporary query engines do exploit in practice).

Flummi's core approach and the use of SQL as compilation target can be pursued much further. A multitude of loose ends are waiting to be picked up, among these a notion of parallelism in the spirit of Unix' `fork` in which one basic block outputs $n \geqslant 2$ `JUMP` rows to spawn n *independent strands of computation* which rejoin only later. We hypothesize that *recursive programs* can be understood in a quite similar fashion.

6 ACKNOWLEDGMENTS

We thank Thomas Neumann for his quick and insightful responses to our questions regarding the *Umbra* relational database system. This research has been supported by the DFG under grant no. *GR 2036/6-1*.

REFERENCES

[1] F.E. Allen. 1970. Control Flow Analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.
[2] N. Armenatzoglou, S. Basu, N. Bhanoori, and M. Cai. 2022. Amazon Redshift Re-invented. In *Proc. SIGMOD*. Portland, OR, USA.
[3] F. Bancilhon. 1986. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*. Springer, 165–178.
[4] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *Proc. CIDR* (Santa Cruz, CA, USA).
[5] G.E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996).
[6] T. Burghardt, D. Hirn, and T. Grust. 2022. Functional Programming on Top of SQL Engines. In *Proc. PADL*. 59–78.
[7] A. Cheung, A. Solar-Lezama, and S. Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.
[8] E. Cooper, S. Lindley, Philip W., and J. Yallop. 2006. Links: Web Programming Without Tiers. In *International Symposium on Formal Methods for Components and Objects*. 266–296.
[9] C. Duta. 2022. Another Way to Implement Complex Computations: Functional-Style SQL UDF. In *Proc. HILDA*. 1–7.
[10] C. Duta and T. Grust. 2020. Functional-Style UDFs With a Capital 'F'. In *Proc. SIGMOD*. Portland, OR, USA.
[11] A. Eisenberg and J. Melton. 1999. SQL:1999, Formerly Known as SQL3. *ACM SIGMOD Record* 28, 1 (March 1999).
[12] K.V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proc. SIGMOD*. San Francisco, CA, USA.
[13] S.J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. 1996. Expressive Recursive Queries in SQL. Joint Technical Committee ISO/IEC JTC 1/SC 21 WG 3, Document X3H2-96-075r1.
[14] T. Fischer, D. Hirn, and T. Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *Proc. SIGMOD*. New York, NY, USA, 23892392.
[15] Y. Foufoulas, A. Simitsis, L. Stamatogiannakis, and Y. Ioannidis. 2022. YeSQL: "You Extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB* 15, 10 (jun 2022), 22702283.
[16] K. Franz, S. Arch, D. Hirn, T. Grust, T.C. Mowry, and A. Pavlo. 2024. Dear User-Defined Functions, Inlining Isnt Working Out So Great for Us. Lets Try Batching To Make our Relationship Work. Sincerely, SQL. In *Proc. CIDR*. Chaminade, CA, USA.
[17] T. Grust, J. Rittinger, and T. Schreiber. 2010. Avalanche-Safe LINQ Compilation. *Proc. VLDB* 3, 1-2 (2010), 162–172.
[18] S. Gupta, S. Purandare, and K. Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proc. SIGMOD*. Portland, OR, USA.
[19] S. Gupta and K. Ramachandra. 2021. Procedural extensions of SQL: understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (apr 2021), 13781391.
[20] R. Guravannavar and S. Sudarshan. 2008. Rewriting Procedures for Batched Bindings. *Proc. VLDB* 1, 1 (2008).
[21] D. Hirn. 2023. Data is Data and Control Should be Data, Too. In *Proc. VLDB PhD Workshop*. Vancouver, Canada.
[22] D. Hirn and T. Grust. 2020. PL/SQL Without the PL. In *Proc. SIGMOD*. Portland, OR, USA.
[23] D. Hirn and T. Grust. 2021. One WITH RECURSIVE Is Worth Many GOTOs. In *Proc. SIGMOD*. Xian, Shaanxi, China.
[24] Holtsetio. 2019. MySQL Raytracer. `https://demozoo.org/productions/268459/`.
[25] R.A. Jarvis. 1973. On the Identification of the Convex Hull of a Finite Set of Points in the Plane. *Inform. Process. Lett.* 2, 1 (1973), 18–21.
[26] V. Leis, P. Boncz, A. Kemper, and T. Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proc. SIGMOD*. Snowbird, Utah, USA, 743754.
[27] Logica [n.d.]. *Logica*. `https://logica.dev/`.
[28] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 4 (1976), 308–320.
[29] F. McSherry. 2022. Materialize: A Platform for Building Scalable Event-Based Systems. In *Proc. DEBS*. Copenhagen, Denmark.
[30] E. Meijer, B. Beckman, and G. Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .Net Framework. In *Proc. SIGMOD*. Chicago, IL, USA, 706–706.
[31] T. Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB* 4, 9 (2011).

[32] T. Neumann and M.J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proc. CIDR*. Amsterdam, The Netherlands.
[33] T. Neumann and A. Kemper. 2015. Unnesting Arbitrary Queries. In *Proc. BTW*. Hamburg, Germany, 383–402.
[34] S. Palkar, J.J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *Proc. CIDR*. Chaminade, CA, USA.
[35] M. Raasveldt and H. Muehleisen. [n.d.]. *DuckDB*. `https://github.com/duckdb/duckdb`
[36] K. Ramachandra and K. Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proc. VLDB* 12, 12 (2019).
[37] K. Ramachandra, K. Park, K.V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. 2018. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB* 11, 4 (2018).
[38] L.A. Rowe and M. Stonebraker. 1987. The POSTGRES Data Model. In *Proc. VLDB*. Brighton, UK.
[39] M. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *Proc. SSDBM*. Tampa, FL, USA, 2536.
[40] M. Sichert and T. Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB* 15, 5 (2022), 1119–1131.
[41] SQL:1999 [n.d.]. *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.
[42] G. Steele. 2017. It's Time for a New Old Language. In *Proc. PPoPP*. Austin, TX, US.
[43] G. Zhang, B. Mariano, Xipeng Shen, and I. Dillig. 2023. Automated Translation of Functional Big Data Queries to SQL. *Proc. OOPSLA (SPLASH)* 7 (2023), 580–608.
[44] G. Zhang, Y. Xu, X. Shen, and I. Dillig. 2021. UDF to SQL Translation Through Compositional Lazy Inductive Synthesis. *Proc. OOPSLA (SPLASH)* 5 (2021), 1–26.