Trampoline-Style Queries for SQL

Louisa Lambrecht Torsten Grust
University of Tübingen
Tübingen, Germany
firstname.lastname@uni-tuebingen.de

Altan Birler Thomas Neumann
Technical University of Munich
Munich, Germany
firstname.lastname@tum.de

ABSTRACT

We introduce *trampoline-style queries* as an alternative expressive foundation for iterative computation in SQL. A trampoline repeatedly executes a family of branch queries which can exercise fine-grained control over (1) the routing of rows between iterations and (2) the emission of output rows. We relate trampoline-style queries to the established recursive CTEs, provide a taste of how trampolines elegantly cover a wide range of applications, and shed light on strategies for their massively parallel implementation inside contemporary relational database systems.

1 FROM FIXPOINTS TO TRAMPOLINES

The ability of SQL:1999's `WITH RECURSIVE` [16] to perform general iterative computation over tabular data right inside the query engine should render it most popular among database practitioners. Instead, the construct leads a rather shadowy existence, with a reputation of having obscure semantics and an inefficient implementation.

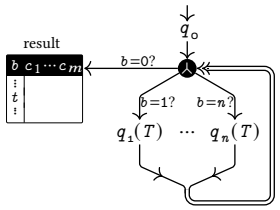
`WITH RECURSIVE` T `AS (`q_0 `UNION ALL` $q_\infty(T)$`)` computes the *fixpoint* of the iterated query q_∞ [6]. With their roots in database and set theory, fixpoints do provide a fitting and expressive foundation for in-database iteration, but they require semi-naive evaluation [1] to be computed efficiently: this mode of query evaluation (1) limits the visibility of already computed results, and (2) relies on monotonicity constraints, enforced through ad-hoc, far-reaching (yet incomplete) syntactic restrictions imposed on the iterated query q_∞. In daily SQL pratice, theses issues lead developers to workarounds that may make queries both hard to read and incur runtime penalties [5].

Trampoline style for SQL. Given these long-standing deficits of `WITH RECURSIVE`, this paper explores the design of an iteration construct for SQL that banks on a different general foundation for iterative computation: *Trampoline style* has its roots in the programming languages community, originally devised as a compilation technique for recursive programs [15]. A trampoline-style program repeatedly executes a dispatcher (or *trampoline*) that determines the next step of computation to be performed. The program's current iteration may emit results as well as directions for the dispatcher on how to proceed in the subsequent iteration. Any iterative computation can be cast into trampoline style [3].

 15th Annual Conference on Innovative Data Systems Research (CIDR'25). January 19-22, 2025, Amsterdam, The Netherlands.

```
1 WITH TRAMPOLINE
2 T(b, c1, ..., cm) BRANCH(b) AS (
3                q0
4   BRANCH 1: q1(T)
5                ⋮
6   BRANCH n: qn(T)
7 )
8 TABLE T;
```

(a) Prototypical form of a `WITH TRAMPOLINE` query.



(b) Sketch of the data flow through `WITH TRAMPOLINE`.

Figure 1: Trampoline style for SQL: `WITH TRAMPOLINE`.

We argue that *trampoline style* can be adapted to serve as an expressive, elegant, and efficient alternative foundation for iteration in SQL. In what follows, we discuss `WITH TRAMPOLINE`, a SQL construct that builds on the trampoline style of computation. In imitation of the original trampoline style, a `WITH TRAMPOLINE` query dispatches its input to one of several branch queries on a row-by-row basis, thus exercising fine-grained control over the computation performed in each iteration (as opposed to `WITH RECURSIVE` which threads all rows through q_∞). Complex iterative and branching data flow maps to trampoline queries in a concise and modular fashion. In consequence, trampoline-style queries can simulate stateful imperative programs, run pattern matchers over masses of time-series data, or help to efficiently implement the cascading semantics of SQL DML statements. We elaborate on these exemplary use cases below.

`WITH TRAMPOLINE`. To make things tangible, let us refer to the protoypical trampoline-style SQL query in Figure 1a[1] and the sketch of its data flow in Figure 1b.

- The initial iteration of this `WITH TRAMPOLINE` query evaluates SQL query q_0, yielding a table T with columns $(b, c_1, \ldots, c_m)$.
- In each subsequent iteration, all queries in the branches $t = 1, \ldots, n$ are run. A dispatcher ensures that query q_t will be evaluated only over those rows directed to its branch t (these are the rows in table T with column $b = t$; in Figure 1b, see the edges from the dispatcher to the $q_1, \ldots, q_n$).
- The branch queries q_t may return rows with column b set to
  - $b = 0$ to indicate that these rows should be collected to form the overall result table (once we collect such a row for output, we set its column b to t to record that branch query q_t emitted that row), and/or

[1] We consider the exact syntactic form of `WITH TRAMPOLINE` to be secondary—the prototypical syntax in Figure 1a should serve us well for now.

– $b \in \{1, \ldots, n\}$ to direct these rows to branch b in the next iteration (if none of the branches emits such rows, evaluation of `WITH TRAMPOLINE` is complete).

The following pages shed light on `WITH TRAMPOLINE`, its semantics, applications, and implementation aspects. We will show how trampolines

- directly express looping programs in SQL which admits the concise purely SQL-based formulation of textbook-style algorithms,
- generalize `WITH RECURSIVE`, suggesting that trampolining is not alien and fits well with existing SQL theory and practice (see Section 2),
- naturally support exemplars of complex iterative computation in SQL, leading to queries that are readable as well as efficient to evaluate (Section 3), and
- lend themselves to an efficient parallel implementation inside modern database kernels (we have integrated `WITH TRAMPOLINE` into Umbra [19], see Section 4).

2 A SPOTLIGHT ON WITH TRAMPOLINE

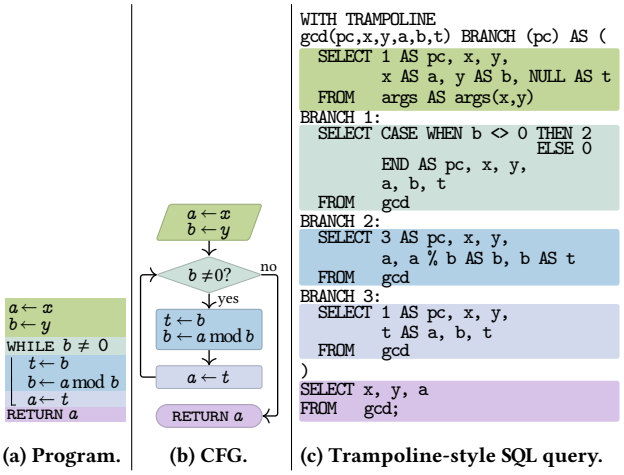
There are several ways to suitably understand the computation performed by `WITH TRAMPOLINE`:

(1) Its **set-oriented operational semantics** generalizes the behavior of `WITH RECURSIVE` (we elaborate on this in Section 2.1).
(2) A trampoline query may act like a **state machine**: Each state is represented by a branch t whose query q_t performs state-specific computation and determines the successor states (Section 3.1 takes this particular view of `WITH TRAMPOLINE`).
(3) `WITH TRAMPOLINE` can **simulate virtual machines (VMs) or threaded interpreters**: branch t represents the VM instruction at program location t. Branch query q_t performs that instruction, possibly emits results, and directs the computation to the following instruction(s).

The latter VM-like view of `WITH TRAMPOLINE` admits the straightforward transcription of imperative programs into SQL. Figure 2 shows how this could look like for a textbook-style *greatest common divisor* algorithm `gcd(x,y)`. From the statements of the program in Figure 2a (or, equivalently, from the nodes of its control flow graph in Figure 2b) we directly derive the branches of the trampoline query in Figure 2c. The branch queries return rows (`pc`, `x`, `y`, `a`, `b`, `t`) in which columns `a`, `b`, and `t` hold the current bindings of the program's variables. Column `pc` decides which branch will execute next and thus acts like a program counter: for example, column expression `1 AS pc` in the `SELECT` clause of `BRANCH 3` directs execution back to `BRANCH 1`, effectively implementing the looping control flow edge in Figure 2b. Likewise, the `CASE` conditional in `BRANCH 1` implements the branching control flow in the CFG of Figure 2b.

Note how *one* evaluation of the `WITH TRAMPOLINE` query simulates n runs of the VM if table `args(x,y)` holds n input rows: Each row in table `gcd` defines the state of one of these VMs which can independently proceed (`pc` $\in \{1, 2, 3\}$) or emit output and halt (`pc` $= 0$) as indicated by their local variable bindings.

The control flow of `gcd` is characterized by a single loop. The direct transcription of imperative code into a trampoline query, however, does apply to looping and branching control flow of arbitrary



(a) Program. (b) CFG. (c) Trampoline-style SQL query.

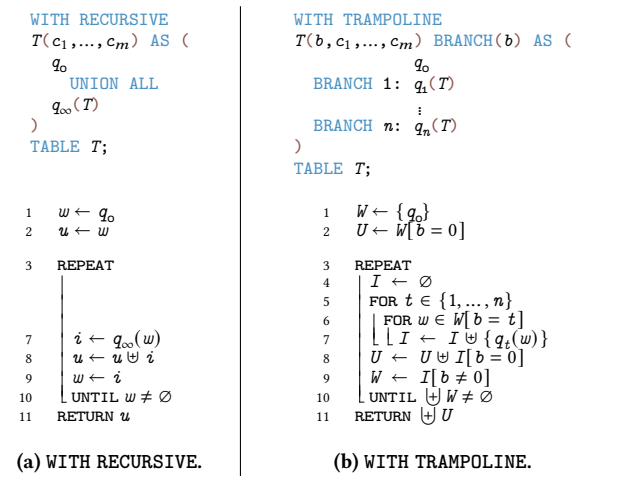
Figure 2: Greatest common divisor: from program to query.

complexity. Indeed, such translations of imperative programs into parallel SQL-based evaluators [3, 18] has been one principal motivation behind the design of `WITH TRAMPOLINE`. It may be well worth revisiting this older work now that trampoline-style queries are available in SQL.

2.1 Set-Oriented Operational Semantics

Despite their disparate roots in query and programming languages, respectively, the operational semantics of `WITH RECURSIVE` and `WITH TRAMPOLINE` align in interesting ways. Figure 3 puts both side by side.

`WITH RECURSIVE` (Figure 3a). Starting with the rows placed by q_0 into *working table* w, `WITH RECURSIVE` evaluates q_∞ over w to yield the *intermediate table* i. The rows in i are added to the overall result table u (the *union table*) and are also made available in working table w to prepare the next iteration of q_∞. Computation stops, once i contributes no further rows [1, 4, 6].



(a) `WITH RECURSIVE`. (b) `WITH TRAMPOLINE`.

Figure 3: Side-by-side comparison of the loop-based, operational semantics for `WITH RECURSIVE` and `WITH TRAMPOLINE`.

`WITH TRAMPOLINE` **(Figure 3b).** The leap to `WITH TRAMPOLINE` trades working table w for a *set of working tables* W (likewise for the intermediate table i and union table u which now become sets of tables I and U, respectively). `WITH TRAMPOLINE` starts out with a singleton working table set $W = \{q_0\}$ and then repeatedly runs the dispatcher in Lines 5–7 to evaluate the branch queries q_t ($t \in \{1, \ldots, n\}$). Branch query q_t is evaluated over all working tables in W but only processes the subset of rows directed to its branch t (we define $R[p]$ to evaluate predicate p over all tables in R as

$$R[p] \equiv \{s \mid r \in R,\ s \coloneqq \sigma[p](r),\ s \neq \varnothing\}$$

and thus can identify the working table subsets relevant for q_t via $W[b = t]$). These independent evaluations of the branch queries yield a set of intermediate tables I. The subsets $I[b \neq 0]$ of these intermediate tables directed to the next iteration are made available as the new working table set W. The table subsets $I[b = 0]$ holding result rows instead are added to U which we ultimately flatten (Line 11) to return one final result table.

The operational semantics do not yet define an efficient evaluation strategy for `WITH TRAMPOLINE`, but (1) the iteration over the n branches (see Line 5) as well as (2) the independent evaluation of the q_t over the separate working tables (Lines 6–7) are obvious sources of parallelism. We explore this is in Section 4 below.

`WITH TRAMPOLINE` **generalizes** `WITH RECURSIVE`**.** Trampoline-style CTEs generalize regular recursive CTEs: the trampoline query of Figure 4 computes the same result as the `WITH RECURSIVE` CTE shown in Figure 3a. In Figure 4, T_{01} denotes a table with single column b holding two rows with values 0 and 1. Every iteration of this trampoline query evaluates q_∞ whose rows are added to the union table ($b = 0$) *as well as* the working table ($b = 1$), the latter of which will be fed back to q_∞ in the subsequent iteration. In terms of the operational semantics of Figure 3b, since we iterate a single branch, I and W remain singleton table sets. While we do not propose to replace `WITH RECURSIVE`, this semantic affinity of both iterative constructs suggests that (a) `WITH TRAMPOLINE` is not as alien as its PL roots may suggest and (b) that existing database kernel infrastructure can be carried over or adapted to support trampline queries.

```
WITH TRAMPOLINE
T(b,c1,...,cm) BRANCH(b) AS (
            SELECT T01.b, q0.*
            FROM q0, T01
 BRANCH 1: SELECT T01.b, q∞.*
            FROM q∞(T), T01
)
SELECT c1,...,cm
FROM   T;
```

Figure 4: `WITH TRAMPOLINE` can mimic `WITH RECURSIVE`.

3 WHAT `WITH TRAMPOLINE` CAN DO FOR YOU

`WITH TRAMPOLINE` is a versatile iteration construct with a wide range of applications. Below we discuss two scenarios that aim to highlight the efficiency, expressiveness, and elegance of trampoline-style SQL queries.

3.1 Row Pattern Matching

There is an immediate correspondence between *finite state machines* (FSMs) and trampoline-style queries in which each FSM state maps to its dedicated branch in the `WITH TRAMPOLINE` CTE. Here,

gas_prices

seq	timestamp	price	⋯	definition
1	2019-01-02 08:15:06	1609	⋯	NULL
2	2019-01-02 12:10:07	1569	⋯	BOTTOM (∨)
3	2019-01-02 12:18:06	1579	⋯	UP (/)
4	2019-01-02 14:36:07	1583	⋯	UP (/)
⋮	⋮	⋮	⋮	⋮

Figure 5: Tabular input for row pattern matching: gas prices over time (rows ordered by `seq`, tagged by `definition`).

we build on this correspondence to perform *row pattern matching* in the style of `MATCH_RECOGNIZE`, a still widely unsupported construct that has recently been added to the SQL standard [17]. The resulting trampoline query massively parallelizes the pattern matching process and executes significantly faster than dedicated implementations of `MATCH_RECOGNIZE`.

Row pattern matching with `MATCH_RECOGNIZE`. Row pattern matching evaluates regular expressions over the rows of an ordered input table. Table `gas_prices` of Figure 5, in which column `seq` orders rows based on `timestamp`, would be typical input.[2] Column `definition` tags rows to identify current downward/upward trends (`DOWN`/`UP`) or local minima (`BOTTOM`) in gas prices. Below, we abbreviate these row tags—or *definitions* in the parlance of `MATCH_RECOGNIZE`—by \, /, and ∨. Given this ordered and tagged input table, regular expression $re = \neg(\backslash^*)\vee(/^*)\llcorner$ over column `definition` identifies sequences of rows that represent temporary gas price drops ¬⌟. Beyond such regular row patterns, `MATCH_RECOGNIZE` can refine pattern matches based on *measures* that are recorded while rows are processed. We disregard measures here to maintain focus.

Simulating FSMs using `WITH TRAMPOLINE`. Figure 6 casts regular expression re into a four-state FSM. We can directly map this state machine into a `WITH TRAMPOLINE` query in which (1) initial query q_0 implements the FSM's start state 0 and (2) branch query q_t realizes state t, $t \in \{1, 2, 3\}$. Branches 1 and 3 of the resulting trampoline query are reproduced in Figure 7. The CTE computes table `fsm(state,seq,`⟨*measures*⟩`)` in which column `state` encodes the current FSM state. Branch queries typically perform state-specific computation—maintain measures, for example—and then consume the next row from the input table: query q_1 in branch 1 inspects the row tag `g.definition` $\in$ {`'\'`, `'∨'`} to emit rows with `state` $\in \{1, 2\}$ in order to transition to the correct subsequent state. (Aside: q_1 uses `CASE...WHEN` to realize this forking transition in FSM state 1. We could alternatively employ the SQL expression `1+(g.definition='∨')::int AS state`—such "*computed GOTOs*" are generally useful idioms in `WITH TRAMPOLINE` queries.)

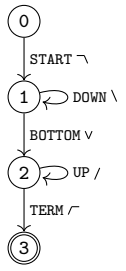


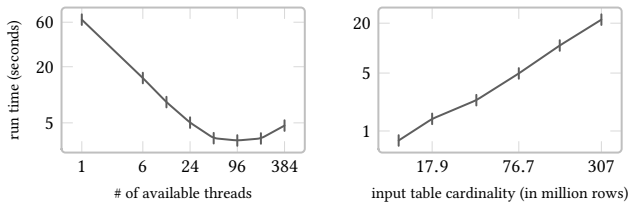
Figure 6: FSM to detect ¬⌟.

Massively parallel FSM runs. This trampoline-based simulation of FSMs can implement high-throughput row pattern matchers. To make this point, we feed a `gas_prices` table of 307 million rows

[2] Table `gas_prices` has been derived from the *Tankerkönig* data set available at `https://creativecommons.tankerkoenig.de/`.

```
1  WITH TRAMPOLINE
2  fsm(state,seq,⟨measures⟩) BRANCH (state) AS (
3   ⋮
4  BRANCH 1: -- q1: branches to state 1 or 2
5    SELECT CASE g.definition WHEN '\' THEN 1 -- DOWN: to state 1
6                             WHEN 'V' THEN 2 -- BOTTOM: to state 2
7           END AS state, g.seq, ⟨maintain measures⟩
8    FROM   fsm JOIN gas_prices g ON fsm.seq+1 = g.seq
9    WHERE  g.definition IN ('\', 'V')
10  ⋮
11 BRANCH 3: -- q3: accepting state, emit match
12   SELECT 0 AS state, seq, ⟨finalize measures⟩
13   FROM   fsm
14 )
15 SELECT seq AS match, ⟨measures⟩ -- output row pattern matches
16 FROM   fsm;
```

Figure 7: `WITH TRAMPOLINE` query implementing the FSM of Figure 6 (excerpt, only branches/states `1` and `3` shown).



(a) Trampoline run time for 30 million row pattern matches.

(b) Full run time for varying input cardinalities (at 96 threads).

Figure 8: Run times for trampoline-based `MATCH_RECOGNIZE`.

into a `WITH TRAMPOLINE` query that detects the ⌝⌜ pattern of Figure 6. Once initial query q_0 has identified the rows whose `START` (⌝) tag indicate the start of a potential pattern match, about 30 million rows remain in table `fsm` and enter the iterative data flow. Each such row defines one FSM run and, effectively, *the trampoline thus pursues 30 million FSM runs in parallel*. Runs independently transition between states based on the rows read from the `gas_prices` table; runs are dropped as soon as they are found to be rejected by the FSM: the length of a match determines how many iterations a row is kept in table `fsm`. For our example, the trampoline performs an average (maximum) number of 3.8 (29) iterations per match.

This massively parallel FSM simulation benefits if the host offers multiple execution threads (see Figure 8a which shows run times recorded on a computer with two AMD EPYC™ 7402 CPUs with 24 cores/48 threads per CPU): using 96 threads, the trampoline completes the 30 million FSM runs within 3.17 seconds (at an input consumption rate of 9 580 585 rows/second). Given this row pattern matching core based on `WITH TRAMPOLINE`, Umbra can run the full `MATCH_RECOGNIZE` query (including row tagging and derivation of measures) over an input table of 307 million rows in about 20 seconds (Figure 8b). In comparison, Trino's [11] dedicated implementation of `MATCH_RECOGNIZE` requires 566 seconds (25× longer) on the same host.

3.2 Cascading Deletes

Database systems have to preserve referential integrity: when a row is deleted, all rows that reference it with a `CASCADE` foreign key constraint must be deleted as well. The graph formed by such foreign key relationships can be quite complex and possibly contain cycles, implying the need for iterative deletion processing. Consider the tables (nodes) and foreign key relationships (edges) in Figure 9 for the LDBC Social Network Benchmark [9]. This benchmark simulates a social network of people, forums, and messages. When a message is deleted, all replies to that message, and replies to those replies must be deleted recursively. Likewise, when a person is deleted, forums moderated by the person, messages within those forums, and messages directly created by that person must all be deleted.

Deletion of rows happens in two phases to prevent the "Halloween Problem" [20]. First, the rows to be deleted are collected. Second, the rows are physically deleted after *all* affected rows are collected, preventing the physical deletions from interfering with the scans producing the rows. The first phase is the most challenging part of cascading deletion. While most database engines rely on a specialized implementation for row collection, the `WITH TRAMPOLINE` construct provides an elegant and efficient alternative. For every row deleted from a table, we need to join with referencing tables and delete the matching rows. To facilitate this, we define a branch query per table and relationship, where *table branches* forward rows to connected *relationship branches*. *Relationship branches*, in turn, perform joins to determine rows in connected tables. The structure of the `WITH TRAMPOLINE` query thus is *statically determined by the current database schema*: whenever a user requests to delete rows from a table, the corresponding `WITH TRAMPOLINE` query can be automatically generated and executed by the database.

Figure 10 shows the skeleton of a `WITH TRAMPOLINE` construct that collects the row IDs (columns `rid`) of all affected rows when the person with ID `56` is deleted. In a post-processing step, the row IDs collected by the *table branches* may then be used to physically delete the actual rows in a bulk fashion. Note that a *table branch* may need to forward rows to multiple *relationship branches* if the table has multiple foreign key relationships. This is the case for the `Person` table, which is connected to the `Forum` and `Message` tables over the `moderates` and `creates` relationships. In our implementation, forwarding rows to multiple branches is achieved by using an implicit cross product with the target branches as seen in Line 7 of Figure 10.

There are multiple potential advantages to using the `WITH TRAMPOLINE` construct for cascading deletes over a special implementation relying solely on index traversal:

- Large and complex cascading deletes can be handled efficiently with a scalable implementation of trampolines as described in Section 4.
- Using a generic query with trampoline and join operators allows a system to utilize its query optimizer to do proper physical operator selection, leading to more efficient execution for a larger

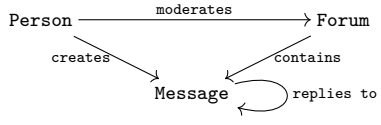


Figure 9: Simplified schema of the LDBC Social Network with cyclic foreign key constraints.

```
WITH TRAMPOLINE
TBD(b, rid) BRANCH(b) AS (
  -- initialize: delete Person with row id 56
  SELECT 'Person' AS b, 56 AS rid
BRANCH 'Person':
  SELECT e AS b, TBD.rid
  FROM   TBD, (VALUES (0), ('moderates'), ('creates')) AS edge(e)
BRANCH 'Forum':
  SELECT e AS b, TBD.rid
  FROM   TBD, (VALUES (0), ('contains')) AS edge(e)
BRANCH 'Message':
  SELECT e AS b, TBD.rid
  FROM   TBD, (VALUES (0), ('replies to')) AS edge(e)
BRANCH 'moderates': -- Person -> Forum
  SELECT 'Forum' AS b, f.rid
  FROM   TBD JOIN Forum f ON f.moderator_rid = TBD.rid
BRANCH 'creates': -- Person -> Message
  SELECT 'Message' AS b, m.rid
  FROM   TBD JOIN Message m ON m.creator_rid = TBD.rid
BRANCH 'contains': -- Forum -> Message
  SELECT 'Message' AS b, m.rid
  FROM   TBD JOIN Message m ON m.container_rid = TBD.rid
BRANCH 'replies to': -- Message -> Message
  SELECT 'Message' AS b, m.rid
  FROM   TBD JOIN Message m ON m.reply_to_rid = TBD.rid
)
-- Now perform bulk deletion
-- TBD.b holds the source branch (the table to delete from)
PERFORM delete_row(TBD.b, TBD.rid)
FROM    TBD;
```

Figure 10: Trampoline-based deletion for the person with row ID 56, cascading to the corresponding forums and messages. Here, branches are identified by the values of an `ENUM` type rather than integers. Rows to be deleted are forwarded to the branches `'Person'`, `'Forum'`, and `'Message'`, which in turn output the rows. All outputted rows are then deleted in a post-processing step using the *delete_row* function.

class of delete queries. For example, in the LDBC dataset, the deletion of a single person could result in the deletion of millions of messages. Still, one would expect most deletions to be relatively small, *e.g.*, deleting a single message or a single person with few messages and replies. This implies that the selection of physical operators can be crucial for the performance of deletes.

- An alternative duplicate-eliminating set-based semantics for trampolines can be utilized to prevent rows from being collected for deletion multiple times. Recursive CTEs can be executed with `UNION` or `UNION ALL` semantics, where `UNION` automatically removes and avoids redundant processing of duplicate rows. Nonetheless, the semantics of recursion in SQL is all or nothing, *i.e.*, either duplicates must be removed for the entire recursion (which can be costly) or not at all. `WITH TRAMPOLINE` can be extended for a more fine-grained control over the removal of duplicates: individual branches could be marked as *duplicate-eliminating* without impact on the rest of the trampoline.

4 IMPLEMENTING PARALLEL TRAMPOLINING

While the semantics of `WITH TRAMPOLINE` is not too complex, providing an efficient implementation in terms of a query engine operator is challenging. The variety of possible use cases is huge: some queries have many branch queries actively producing rows, some just a few. Some branch queries produce only a few rows per invocation, others produce thousands of rows. Sometimes most

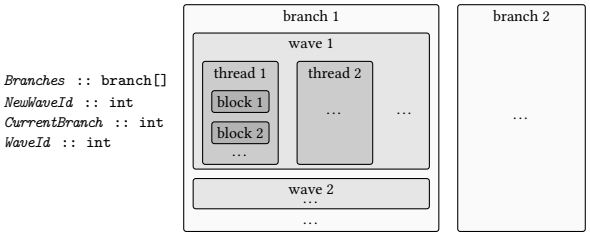


Figure 11: Data structures used by Umbra's trampoline.

rows target the same branch, sometimes rows are spread across all branches. Combined with multi-threading, this makes a scalable implementation challenging. Below, we sketch the implementation of the trampoline operator that has been integrated into the Umbra relational database system [19].

Conceptually, we want to process all rows that are in flight in the trampoline operator in parallel. However, there are two constraints to that. First, we want to avoid updating shared data structures as much as possible for performance reasons. And second, the query semantics do not allow for arbitrary parallelization: as shown in Figure 3, the operator conceptually evaluates the branch queries q_t on a collection of individual working tables $w \in W[b = t]$. If branch query q_t is *non-linear* [8] (*e.g.*, because it contains a `GROUP BY`), we must evaluate the query over each working table w in separation. In the implementation we call that a *wave*: a wave consists of all rows returned by a branch that have to be evaluated together. Within a wave, rows are collected in thread-local regions, which allows threads to add rows to a wave without any kind of locking. This architecture is shown in Figure 11: Each trampoline branch contains a list of waves, each wave consists of a list of thread-local data structures, and each of these contains lists of rows that are organized in blocks of up to a thousand rows. We use this block mechanism to ensure good thread utilization; in a query, a single thread might produce thousands of rows for a given branch, while all other threads end up producing rows for other branches. When reading these rows again in the next trampoline iteration, we can use the block organization to distribute rows across threads via work stealing.

Importantly, we do not have to maintain wave boundaries if *all branch queries are linear* [8]. For a linear branch query, we have

$$q_t(w_1) \cup \cdots \cup q_t(w_n) = q_t(w_1 \cup \cdots \cup w_n) \ .$$

(All branch queries shown in this paper happen to be linear.) Merging the w_i in this fashion is highly desirable as it allows much stronger parallelism while evaluating q_t. The system thus will try to fuse waves if possible. Umbra tracks the linearity of query operators and can detect when the path from the trampoline operator's source (which reads the current wave) to the trampoline sink (which assigns rows to their target branch in the next iteration) exclusively consists of operators that are linear in the input containing the trampoline source.

Trampoline sink and source in the consume/produce model. As meta-information the trampoline operator maintains (1) the *CurrentBranch* currently being read, (2) the *WaveId*, a monotonic growing number describing the current wave that is read from the current branch, and (3) the *NewWaveId*, which is the wave id

```
fun consumeTrampolineSink(row):
  br = Branches[row.targetBranch]
  # check if we are inserting into a different wave
  w = last wave in br
  if w.waveId != NewWaveId:
    # check if the waves can be merged
    if CurrentBranch != row.targetBranch and
       allBranchesLinear and w.waveId > WaveId:
      w.waveId = NewWaveId
    else
      # create correct wave in target branch
      w = create new wave NewWaveId in br
  # locate per-thread data within wave
  t = create or access per-thread data in w
  # store row in current block
  b = last block in t
  if b is full:
    b = allocate new block in t
  store row in b
```

Figure 12: Pseudo code of the trampoline sink.

```
fun produceTrampolineSource():
  br = Branches[CurrentBranch]
  # read the current wave
  w = br.waves[WaveId]
  if w has more blocks:
    return next block from w
  # wave finished, execute operators above the source
  finalize current branch br
  NewWaveId += 1
  drop wave from current branch br
  # prefer more waves from the same branch
  if br has more waves:
     WaveId = min waveId in br
     return produceTrampolineSource()
  # the current branch has no more data, switch branches
  CurrentBranch = min branch with waves
  if Branches[CurrentBranch] has waves:
     return produceTrampolineSource()
  return nil
```

Figure 13: Pseudo code of the trampoline source.

of the result wave. With that information, the pseudo code for the *trampoline sink* part of the operator is shown in Figure 12.

The routine first retrieves the selected target branch from the current row and then uses an atomic read to check if the latest wave on that branch is our target wave. If not, it first checks whether it is safe to merge the existing wave with the new wave. If yes, it updates the wave id, otherwise it creates a new wave. Within the wave a thread-local data structure is created or accessed, which renders all further operations lock free. The current row is appended to the thread-local structure, creating new blocks as needed to avoid overly large blocks.

The source side of the trampoline operator picks blocks from the current wave until the wave is exhausted, and then switches to the next wave. The pseudo code is reproduced in Figure 13.

We first produce more row blocks from the current wave until the wave is exhausted, each thread picking blocks on demand. When no more blocks are available, the branch is *finalized*, which means that all operators on the current branch between the trampoline source and the trampoline sink are executed as needed (*e.g.*, a `GROUP BY` result might be computed and then propagated further up). Once we have to switch waves, the routine prefers processing a new wave from the same branch over switching branches in order to maximize query code locality. If no more waves exist, the code switches to the first branch that has an active wave and stops if no such branch exists. Note that the code that picks the next branch has to be written carefully, as a query might have thousands of branches and a naive scan over all branches would be inefficient. It is advisable to implement some form of priority queue to quickly identify the branches with pending waves.

5 EARLIER AND FUTURE WORK

As we write this, we are looking back at 25 years of iterative computation in SQL: `WITH RECURSIVE` was introduced in SQL:1999 [16], following proposals that date back to the mid-1990s [6]. Still, recursive CTEs are probably best described as the "black sheep" of SQL, with a rather exotic semantics, ad-hoc syntactic restrictions, and often disappointing runtime performance. This led developers to resent the construct [5] and, instead, perform iteration outside the database engine core, accepting the inherent cost of data movement and (de-)serialization. Given a steep rise of workloads that are data-*as well as* computation-heavy [2], we argue that in-core iteration in SQL deserves closer scrutiny again.

Variations of the vanilla `WITH RECURSIVE` recipe have already been explored with `WITH ITERATIVE`, a variant that entirely foregoes the maintenance of—a possibly sizable—union table u (see Figure 3a) and instead returns the final intermediate table i. These non-accumulating semantics led to significant runtime and space savings for in-database implementations of clustering, for example [13]. Two further variants of `WITH ITERATIVE`, coined `TTL` and `KEY`, instead exercise fine-grained control over the limited retention or replacement of rows in the union table [4]. `WITH ITERATIVE KEY`, in particular, paves an alternative way towards the expression and execution of imperative algorithms directly in SQL (recall our discussion of `gcd` and Figure 2). Likewise, DBSpinner [7] discusses iterative CTEs that maintain rows in the union table using an *upsert*-based semantics.

RaSQL [10] and Datalog° [12] stick with the original principles of recursion in SQL, but specifically address how iteration can efficiently interplay with grouping and aggregation (a notorious stumbling stone since 1999).

`WITH TRAMPOLINE`'s focus on a single loop that repeatedly dispatches rows has been inspired by the original work on *trampolined style* [15]. There, trampolines were also used to express interleaved threads, parallelism at different granularities, or interruptible and resumable computation. We are positive that adaptions of these carry over to trampoline-style SQL, offering a rich toolbox for the expression of complex computation inside the database engine.

Where we hop next. Beyond a re-evaluation of compilation techniques [3] that map complex imperative programs into iterative SQL queries (recall Section 2), our study of the `WITH TRAMPOLINE` clause itself as well as its efficient implementation is still underway. Regarding semantics, we explore *per-branch duplicate elimination* (in the style of `DISTINCT ON`) which enables selected branches to memoize earlier intermediate results (Section 3.2). Regarding implementation, we consider scheduling strategies that dynamically pick a new current branch based on the count of its in-flight rows

(if all branches are linear, large row counts facilitate parallelism—otherwise, branches of low cardinality may be preferable). In addition, we currently have DuckDB [14] on our workbenches and will integrate trampolining based on the ideas sketched in Section 4.

Acknowledgments. The Tübingen team has been supported by the DFG under grant no. *GR 2036/6-1*.

REFERENCES

[1] F. Bancilhon. 1986. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems*. Springer, 165–178.
[2] M. Boehm, A. Kumar, and J. Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool.
[3] D. Hirn and T. Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *Proc. SIGMOD*.
[4] D. Hirn and T. Grust. 2023. A Fix for the Fixation on Fixpoints. In *Proc. CIDR*.
[5] C. Duta. 2022. Another Way to Implement Complex Computations: Functional-Style SQL UDFs. In *Proc. HILDA*.
[6] 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.
[7] S. Floratos, A. Ghazal, J. Sun, J. Chen, and X. Zhang. 2021. DBSpinner: Making a Case for Iterative Processing in Databases. In *Proc. ICDE*.
[8] G. Moerkotte. 2020. Building Query Compilers. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`.
[9] G. Szárnyas, J. Waudby, B. A. Steer, D. Szakállas, A. Birler, M. Wu, Y. Zhang, P. A. Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB* (2022).
[10] J. Gu, Y.H. Watanabe, W.A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-Aggregate-SQL on Spark. In *Proc. SIGMOD*.
[11] K. Findeisen. 2021. Row pattern recognition with MATCH_RECOGNIZE. `https://trino.io/blog/2021/05/19/row_pattern_matching.html`.
[12] M.A. Khamis, H.Q. Ngo, R. Pichler, D. Suciu, and Y.R. Wang. 2022. Datalog in Wonderland. *ACM SIGMOD Record* 51, 2 (2022).
[13] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. 2017. SQL- and Operator-Centric Data Analytics in Relational Main-Memory Databases. In *Proc. EDBT*.
[14] M. Raasveldt and H. Mühleisen. 2020. Data Management for Data Science: Towards Embedded Analytics. In *Proc. CIDR*.
[15] S.E. Ganz and D.P. Friedman and M. Wand. 1999. Trampolined Style. In *Proc. ICFP*.
[16] SQL:1999 [n.d.]. *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.
[17] SQL:2016 [n.d.]. *SQL:2016 Standard. Database Languages–SQL–Part 5: Row Pattern Recognition in SQL*. ISO/IEC 9075-5:2016.
[18] T. Fischer. 2023. To Iterate Is Human, to Recurse Is Divine — Mapping Iterative Python to Recursive SQL. In *Proc. BTW*.
[19] T. Neumann and M.J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proc. CIDR*.
[20] Tandem Database Group. 1987. NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL. In *Proc. HPTC*.