PgCuckoo: Laying Plan Eggs in PostgreSQL's Nest

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

ABSTRACT

We demonstrate how to use PostgreSQL's *planner hook* to open a side entrance through which we can pass plan trees for immediate execution. Since this reaches deep into PostgreSQL, we implement plan detail inference and decoration to ensure that externally crafted trees perfectly mimic regular plans. Plan trees may then (1) be generated by external code generators that want to use PostgreSQL as a reliable and efficient back-end for new (maybe even non-relational) languages, or (2) stem from experimental rewrites of SQL plans that PostgreSQL itself does not implement (yet). The demonstration provides a live account of what becomes possible once we let PostgreSQL hatch foreign plan eggs.

ACM Reference Format:
Denis Hirn and Torsten Grust. 2019. *PgCuckoo*: Laying Plan Eggs in PostgreSQL's Nest. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320211

1 POSTGRESQL HATCHES FOREIGN EGGS

From its inception as *Berkeley Postgres* in 1986 to this day, the relational database management system PostgreSQL has always been an ideal vehicle for database research and education. The system's internals are open for inspection not only through the user-facing SQL `EXPLAIN` facility, but also through an extensive logging mechanism, pretty printers for syntax, plan, and executor trees, and a variety of performance counters whose current values are exposed in queryable form. Beyond read-only observation, essential aspects of PostgreSQL may be extended or replaced. This begins with an updatable catalog of data types, built-in operators, or access methods, and ends with user-supplied code that be may be loaded and invoked at database engine runtime [6, 11].

PostgreSQL's *hooks* are documented sparingly and are less widely known but reach particularly deep into the system. Hooks represent global code pointers which—once set—the database kernel uses to invoke user-contributed C functions. Passed to and from these functions are data structures that allow to inspect and modify the RDBMS's state as it runs. Since PostgreSQL version 8.3 more than 20 such hooks have been provisioned, allowing to intercept or alter a wide variety of system behaviors, *e.g.*, when users are authenticated, tables or indexes are looked up in the catalog, queries are parsed, plans are explained, access paths selected, joins reordered, or before (or after) a query is executed [11, § H.4].

PgCuckoo. This demonstration builds on the so-called *planner hook* to significantly alter PostgreSQL's operation: we use the hook to inject query *plan trees from outside the system* and have these foreign plans be executed by the system's query executor—like a cuckoo lays an egg in a victim bird's nest. Much like the avian parasite, we need to make sure that our plans perfectly mimic the PostgreSQL originals. Once we achieve that a number of exciting avenues open up:

- Given an **external code generator** for a foreign (maybe even non-relational) query or data processing language, we may count on PostgreSQL as a runtime and execution back-end for that language (Section 2.1).
- We can **stitch together several plan pieces** to fully control the evaluation of subqueries (or query parts, in general) in a fine-grained fashion (Section 2.2).
- We may **improve original plan trees** through rewriting strategies—expressed on the surface query level as well as on plan trees themselves—that are not present in PostgreSQL itself (Section 2.3).
- We may quickly **retrieve canned "plan favorites"** based on the original SQL query text and other system or environment parameters, foregoing costly (and sometimes unpredictable) planning from scratch (Section 2.4).

The underlying PostgreSQL extension, dubbed *PgCuckoo*, is made available at `http://db.inf.uni-tuebingen.de/PgCuckoo` and we encourage everyone to peruse it as they see fit.

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320211

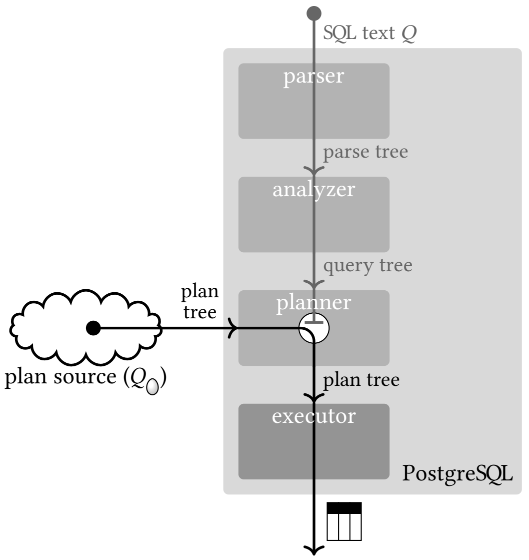


Figure 1: *PgCuckoo* uses PostgreSQL's *planner hook* to inject plan trees from external sources.

2 A DEMONSTRATION OF POSTGRESQL'S PLANNER HOOK

PostgreSQL invokes its *planner hook* [12, file `planner.c`] just before query planning begins. The called user code receives a representation of a parsed SQL query Q and is expected to return a *plan tree* for Q. Plan trees are designed to be *complete* in the sense that they carry all information needed to drive the system's query executor.

We build on this crucial completeness property and take the liberty to return a plan tree that describes the evaluation of an, in general, entirely different query $Q_\bigcirc$ of our choosing. As long as this plan tree is correct and complete—a significant challenge that we address in Section 2.1 below—PostgreSQL's executor will duly evaluate $Q_\bigcirc$ and return its tabular result instead. We thus "short-circuit" the standard planner and effectively close down PostgreSQL's query front end—the executor does not depend on it (symbolized by ⊖ in Figure 1).

This work explores the opportunities that arise when we plug in different PostgreSQL-external plan sources. The live demonstration will consider four such sources, all described below. A wider variety is conceivable.

2.1 External Code Generation

The *planner hook* opens a side entrance for plan trees that we can use to operate PostgreSQL as an execution-only back-end for foreign front-end languages. We benefit from PostgreSQL's proven executor, table storage, and index support and may focus on the development of the front-end itself.

```
SELECT c.c_name, o.o_orderkey, o.o_orderdate,
       abs(o.o_totalprice -
           SUM(l.l_extendedprice *
               (1 - l.l_discount) * (1 + l.l_tax))) AS deviation
FROM   orders AS o, lineitem AS l, customer AS c
WHERE  o.o_orderkey = l.l_orderkey
AND    o.o_custkey  = c.c_custkey
AND    o.o_orderdate > date '1998-01-01'
GROUP BY o.o_orderkey, o.o_totalprice, o.o_orderdate, c.c_name
HAVING SUM(l.l_extendedprice *
           (1 - l.l_discount) * (1 + l.l_tax)) <> o.o_totalprice
ORDER BY c.c_name, o.o_orderkey;
```

Figure 2: Sample SQL query (plan shown in Table 1).

Overall/Inferred Plan Properties	Query Plan	Seed Properties
ⓈⓈⓈⓈⓈ	`PLANNEDSTMT`	
ⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⓇⓇⓇⓇⓂⓂⓂⓂⓂⓂⓂⓂ	`└GROUP AGGGREGATE`	ⒺⒺⓇⓇⓇⓇⓂⓂ
ⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⓇⓇⓂⓂⓂⓂⓂⓂⓂⓂⓂ	`  └SORT`	ⒺⒺⒺⒺⓇⓇⓂⓂⓂⓂ
ⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⓂⓂ	`    └HASH JOIN`	ⒺⒺⒺⒺⓂⓂ
ⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⒺⓂⓂ	`      ├HASH JOIN`	ⒺⒺⒺⓂⓂ
ⒺⒺⒺⒺⒺⒺⒺⒺⓂ	`      │├SEQ SCAN`	ⒺⒺⓂ
	`      ││└lineitem`	
ⒺⒺⒺⒺⓇⓂⓂ	`      │└HASH`	ⒺⓇⓂ
ⒺⒺⒺⒺⒺⓇ	`      │ └SEQ SCAN`	ⒺⓂ
	`      │  └orders`	
ⒺⓇⓂⓂ	`      └HASH`	ⒺⓇⓂ
ⒺⓂ	`       └SEQ SCAN`	ⒺⓂ
	`        └customer`	

Ⓔ expression (×30) Ⓡ intra-plan reference
Ⓜ meta data, misc. flags Ⓢ schema information (×10)

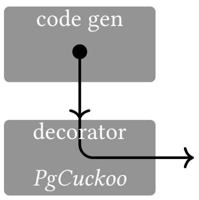
Table 1: Required plan tree properties overall (left), seed from which *PgCuckoo* can infer the rest (right).

Numerous data-centric languages—some non-relational, even—have been developed with PostgreSQL as the execution target in mind. These languages had to be compiled into intermediate SQL code that is then fed into PostgreSQL, often yielding (1) large or non-idiomatic queries that turned out to be challenging for the back-end or (2) queries that failed to fully exploit front-end language semantics. To exemplify, the *Pathfinder* XQuery-to-SQL compiler had perfect information about the (non-)relevance of row order in parts of a generated plan. This knowledge was largely lost during SQL code generation [5]. Further examples of such SQL-emitting systems include *Database-Supported Haskell (DSH)* [13], *Ferry* [4], *GProM* [10], *GraphGen* [14], *Links* [2], *MayBMS* [1], or *Perm* [3].

PgCuckoo, instead, admits the immediate generation of algebraic code in the form of plan trees. Code generators gain full control over plan shape (*e.g.*, type of operations, intermediate materialization, row order preservation, or index usage) and may assemble operator constellations that could never be derived from intermediate SQL code.

Plan trees are PostgreSQL-internal data structures that need to be assembled carefully since the system's executor entirely relies on their completeness and correctness to drive query evaluation. For the moderately complex SQL query shown in Figure 2, the operator nodes of its plan tree already need to be adorned with 2114 parameters and properties, like expression details, schema information, or intra-plan references. The left column of Table 1 shows a breakdown (note that a single 🅔 represents 30 expression details).

To make external plan assembly feasible, *PgCuckoo* provides a *plan decorator* through which code generators may route skeleton plan trees before they are passed into PostgreSQL. The decorator completes the skeleton through rule-based property inference and queries PostgreSQL's catalog to, *e.g.*, infer expression types, resolve operator overloading, and establish node-to-node references. With the plan decorator in place, it suffices to provide about 500 seed properties to formulate the plan above (see the right column of Table 1). All in all, the combination of external code generator and *PgCuckoo*'s plan decorator (shown on the left) assumes the role of the plan source in Figure 1.



The on-site demonstration will exploit that arbitrary plans may be assembled. We will bring a prepared selection of unusual plan trees that do show interesting behavior but would not be considered by PostgreSQL's own SQL compiler (yet, see Section 2.3 below). Additionally, thanks to *PgCuckoo*'s plan decorator, ad-hoc plan trees may be proposed by the audience and then submitted for execution. As long as these plans fulfill basic constraints of consistency, there are countless possibilties to play with *PgCuckoo*.

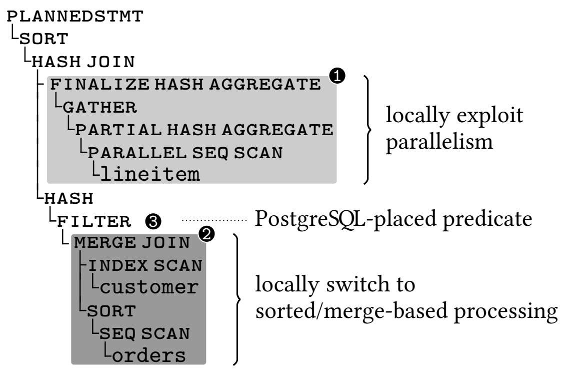
2.2 Fine-Grained Plan Stitching

For the purposes of experimentation, benchmarking, but also in educational settings, it is valuable to be able to exercise precise control over the plan that the RDBMS generates for a SQL query. Vanilla PostgreSQL provides few levers we can pull to influence plan generation. One example are Boolean switches like `set enable_hashjoin = on/off` which govern physical operator choice for the *entire* plan tree [11, § 19.7].

With *PgCuckoo*, we are able to control plan generation at the granularity of individual expressions or subqueries. Several such plan pieces may then be stitched together to form a complete plan for a subject query. We make this functionality available at the surface language level in terms of a table-valued SQL function `plan_execute(·)` whose argument contains a textual representation of the plan piece.

```
SELECT bills.c_name, bills.o_orderkey, bills.o_orderdate,
       abs(bills.o_totalprice - total.price) AS deviation
FROM   plan_execute('{PLANNEDSTMT ❶}') AS total,
       plan_execute('{PLANNEDSTMT ❷}') AS bills,
WHERE  total.l_orderkey = bills.o_orderkey
AND    bills.o_totalprice <> total.price
AND    bills.o_orderdate > date '1998-01-01' -- ❸
ORDER BY bills.c_name, bills.o_orderkey;
```

Figure 3: A rewrite of the query of Figure 2 in which two `plan_execute(·)` calls locally prescribe plans.

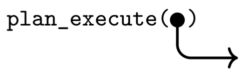


```
PLANNEDSTMT
└SORT
 └HASH JOIN
  ├FINALIZE HASH AGGREGATE ❶
  │└GATHER
  │ └PARTIAL HASH AGGREGATE
  │  └PARALLEL SEQ SCAN
  │   └lineitem
  └HASH
   └FILTER ❸
    └MERGE JOIN ❷
     ├INDEX SCAN
     │└customer
     └SORT
      └SEQ SCAN
       └orders
```

locally exploit parallelism
PostgreSQL-placed predicate
locally switch to sorted/merge-based processing

Figure 4: Stitched plan for the query of Figure 3. The sub-plans ❶ and ❷ have been prescribed by us through `plan_execute(·)`. The residual plan, *e.g.*, the placement of predicate ❸, is decided by PostgreSQL itself.

The evaluation of that plan piece constitutes the function's result, *i.e.*, function `plan_execute(·)` acts as a plan source.



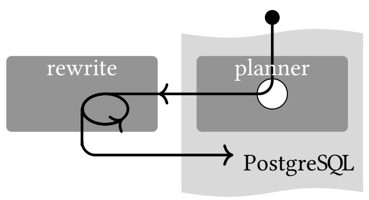
If we place invocations of `plan_execute(·)` at critical or interesting spots in a SQL query, PostgreSQL will generate the global shape of the plan for us while we control plan details locally. Consider the SQL query of Figure 3 in which two `plan_execute(·)` calls prescribe the sub-plans to be used for the evluation of the `total` and `bills` subqueries. In this example, we alter the plan of the original query (Figure 2) to enforce the use of parallelism in plan piece ❶ and to request sort-based (as opposed to hash-based) processing in piece ❷—see the resulting stitched plan in Figure 4 which is a reshaping of the original plan in Table 1.

While such fine-grained stitching facilitates plan manipulation, debugging, and research, we further see its educational value: `plan_execute(·)` provides an ideal tool to study the impact of local planner decisions—the choice of join algorithm or (non-)usage of an index, for example—in the context of real query plans. This includes (more or less obvious) "bad" decisions which are now easily enforced to demonstrate their negative effect. Scenarios of this kind often create the most effective lessons.

For the demo audience, we demonstrate the value of plan stitching with the help of a family of different (yet equivalent) plan fragments that may be "plugged into the holes" of a larger plan tree. This admits the interactive experimentation with sub-plan alternatives that exhibit varying costs, a scenario that would also fit a lab or educational setting well. Since the argument to `plan_execute(·)` is a plain (and thus editable) string, live changes to these fragments may be performed at any time during the demonstration.

2.3 New Plans for SQL

PostgreSQL is a role model of a long-running and well-managed project [6]. An ever-growing legacy of existing applications, however, forces the PostgreSQL community to adopt a conservative development model that preserves "old" expected system behavior. Recent advances in, say, query transformation and optimization thus often only slowly find their way into the system (if at all).



PgCuckoo has been built as a framework in which the experimentation with advanced (or still only half-baked) plan generation approaches may be performed *outside* the database kernel. Thus,

(1) a SQL query Q is submitted to PostgreSQL as usual,
(2) the *planner hook* is used to receive Q's initial plan tree (*PgCuckoo* helps to digest this PostgreSQL-internal data structure, also see the demonstration details below),
(3) an external plan rewriter performs its task, before
(4) the rewritten plan is injected back into PostgreSQL for regular execution.

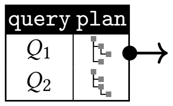
In effect, we obtain an experimental version of the system which explores portions of the plan tree space that an off-the-shelf PostgreSQL would not consider to enter on its own.

The live demonstration showcases an external plan rewriter that implements an adaptation of the advanced and complete SQL query unnesting strategy employed by the *HyPer* DBMS [9]. We further demonstrate how PostgreSQL's plan trees may be abstracted and understood as *logical* relational algebra (for which the unnesting strategy had been designed originally). The audience will witness execution performance improvements up to multiple orders of magnitude.

2.4 Canned Plans for SQL

PgCuckoo has primarily been built to support the ingestion of externally generated plans but also provides a foundation for the construction of *plan caches* or *Query Stores* (Microsoft SQL Server) [8]. Such caches may save repeated plan generation effort and provide predictable performance for queries that are hot-spots in a workload.

In one purely relational design of a plan cache, a table associates a query ID (or hash) Q_i with the best or designated plan tree for that query. When Q_i occurs in the workload, its canned plan tree is retrieved from the cache and passed directly to the executor via the *planner hook*. Alternatively, the cache may hold more than one entry per Q_i to record the query's history of plans and their performance measures. With *PgCuckoo*'s ability to augment this history with plan trees that were *not* hatched up by PostgreSQL itself, all building blocks are in place for an implementation of learning-based plan modification and generation [7] for PostgreSQL.



During the demonstration, a tabular plan cache is available for inspection and experimentation. We bring sample scenarios for a swift first impression but, importantly, the cache may also be manipulated using SQL DML (`INSERT` and `UPDATE`) statements. Writability includes the `plan` column which holds human-readable serialized plan trees: unlike in Microsoft SQL Server's *Query Store*, *PgCuckoo* allows for cached plans to be modified or even created from scratch.

REFERENCES

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. 2008. Fast and Simple Relational Processing of Uncertain Data. In *Proc. ICDE.*
[2] J. Cheney, S. Lindley, and P. Wadler. 2014. Query Shredding: Efficient Relational Evaluation of Queries Over Nested Multisets. In *Proc. SIGMOD.*
[3] B. Glavic and G. Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In *Proc. ICDE.*
[4] T. Grust, J. Rittinger, and T. Schreiber. 2010. Avalanche-Safe LINQ Compilation. In *Proc. VLDB.*
[5] T. Grust, J. Rittinger, and J. Teubner. 2007. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE.*
[6] J.M. Hellerstein. 2019. Looking Back at Postgres. In *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker.* ACM, 205–224.
[7] T. Kraska, M. Alizadeh, A. Beutel, E.H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. 2019. SageDB: A Learned Database Systems. In *Proc. CIDR.*
[8] Microsoft SQL Server [n. d.]. *Microsoft SQL Documentation.* `docs.microsoft.com/en-us/sql`.
[9] T. Neumann and A. Kemper. 2015. Unnesting Arbitrary Queries. In *Proc. BTW.*
[10] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z.H. Liu, V. Krishnaswamy, and V. Radhakrishnan. 2018. Heuristic and Cost-Based Optimization for Diverse Provenance Tasks. *IEEE TKDE* (2018).
[11] PostgreSQL [n. d.]. *The PostgreSQL Relational Database System (Documentation).* `postgresql.org/docs`.
[12] PostgreSQL Git Repository [n. d.]. *Mirror of the Official PostgreSQL Git Repository.* `github.com/postgres/postgres`.
[13] A. Ulrich and T. Grust. 2015. The Flatter, the Better. In *Proc. SIGMOD.*
[14] K. Xirogiannopoulos and A. Deshpande. 2017. Extracting and Analyzing Hidden Graphs from Relational Databases. In *Proc. SIGMOD.*