PgCuckoo – Injecting Physical Plans into PostgreSQL

Denis Hirn[1]

Abstract: Plan forcing is the most capable plan hinting that a database system can implement. It allows the specification of almost every aspect of an execution plan. The open source database system PostgreSQL does not implement any plan hints by default. We will show how an extension can be used to provide plan forcing for PostgreSQL. This extension allows to use the query executor directly and independently of SQL. We sketch some of the interesting use case scenarios for plan forcing in PostgreSQL.

Keywords: PostgreSQL; Physical Algebra; Plan Tree Execution; Query Unnesting

1 Plan Forcing

High-level *declarative* query languages are used to access data stored in a RDBMS. The declarative nature of SQL requires a RDBMS to derive an efficient query evaluation strategy on its own, largely without user guidance. Two key components of the evaluation mechanism of a SQL DBMS are the *query optimizer* and the *query execution engine*. The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and generates an *efficient* execution plan for the given SQL query from the space of possible execution plans. This is a nontrivial task because there can be a large number of possible operator trees for a given SQL query [Ch98].

PostgreSQL uses a disk-aware cost model which combines CPU and I/O costs with certain weights. The cost of a query plan equals the summarized costs of all operators [Le15, 5.1 The PostgreSQL Cost Model]. Using cardinality estimates as its principal input, the query optimizer relies on the cost model to choose the cheapest option from semantically equivalent plan alternatives. Theoretically, as long as the cardinality estimates and the cost models are accurate, this architecture obtains the optimal query plan. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. For real-world data sets, these assumptions are frequently wrong, which may lead to sub-optimal and sometimes disastrous plans [Le15, 1. Introduction].

Plan forcing allows to design and compile physical plans *directly* – there is no SQL or SQL compilation involved. PostgreSQL is well suited for such a far-reaching modification, because it is open source and widely known for its extensibility. Several interesting applications result from the complete control over execution plans provided by plan forcing.

[1] University of Tübingen, Department of Computer Science, denis.hirn@uni-tuebingen.de

- The *planner* is one of the most complex modules of PostgreSQL. This complexity can make it hard to determine whether a new feature or a change to the optimization process may improve query execution performance. Instead of investing time in changing the optimization process, plan forcing can be used to explore portions of the query plan space that the *planner* is unable to enter on its own. Planner functionality that is not available can be simulated this way. In Section 2.1 we will discuss how the textual representation of a plan can be used to alter or create a completely new execution plan. This way, we can manually apply a particular optimization to multiple sample plans to assess the impact on execution performance. Usually it is very hard to construct a plan from scratch, but this process can be simplified significantly, as we will explain in Section 3.2.
- There are plenty of applications that use PostgreSQL's stable execution engine as backend, e.g. for foreign frontend languages. Although some of these tools generate logical or physical algebra internally, intermediate SQL code generation is required because they cannot feed their plans directly into PostgreSQL. Plan forcing can be used to omit this intermediate SQL code and use physical plans instead.
- PostgreSQL's query optimizer is conservative and does not support (advanced) algebraic rewriting of plans. Plan forcing allows to *externalize* system-internal query processing steps. An application of algebraic rewriting is unnesting of *correlated subqueries* as will be explained in Section 3.1. Query unnesting can improve the execution performance significantly but PostgreSQL's planner cannot de-correlate queries in general. Instead of integrating this optimization into PostgreSQL, we can utilize external tools to perform algebraic rewriting. Plan forcing is then used to execute the rewritten plan.
- We can use plan forcing in combination with PostgreSQL's planner by using plan fragments as a part of regular queries. Figure 4 shows how we can incorporate multiple occurrences of the table-valued function `plan_execute` to "stitch together" a plan. But if we use plan forcing for an entire plan we omit PostgreSQL's planner completely, as shown in Figure 3.

2 Physical Plan Injection in PostgreSQL

The logical basis of a RDBMS is usually an extended version of the *relational algebra*, as proposed by Edgar F. Codd [Co70]. An implementation of relational algebra operations is called *physical algebra*. In general, there is no bijective mapping between a SQL query and physical operators because non-trivial SQL queries can be evaluated by numerous semantically identical plans, i.e. constellations of physical operators.

PostgreSQL performs several steps to transform a SQL query from the textual representation into a *plan tree*. The first step involves to *parse* the query, which generates the *parse tree*. Following this, the *analyzer* performs a semantic analysis of the parse tree and creates a

new representation called *query tree*. The *planner* uses the query tree to generate the most efficient plan tree. In the end, the executor evaluates the plan tree. The plan tree is based on a set of data type definitions that represent expressions over the physical algebra. This tree structure serves as the input for the executor. Each node of the tree specifies a physical operator and contains any information needed for its evaluation. These steps are illustrated in Figure 1.

The *parse* and *analyze* modules construct context information in addition to the parse- and query tree. This information is important for the type inference performed in the analyze phase and for the optimization process. The executor does not require any information besides the plan tree of a query, however. This is an important property, because it implies that we can safely skip the *parse*, *analyze* and *optimization* phase without affecting the executor.

To use the PostgreSQL executor independently of SQL, an interface that allows to import, export, and execute plan trees is required. PostgreSQL offers nothing along these lines by default. Still, an extension can be used to implement such an interface. A C language extension can access internal functions and modify the overall behavior of the system [DD03].

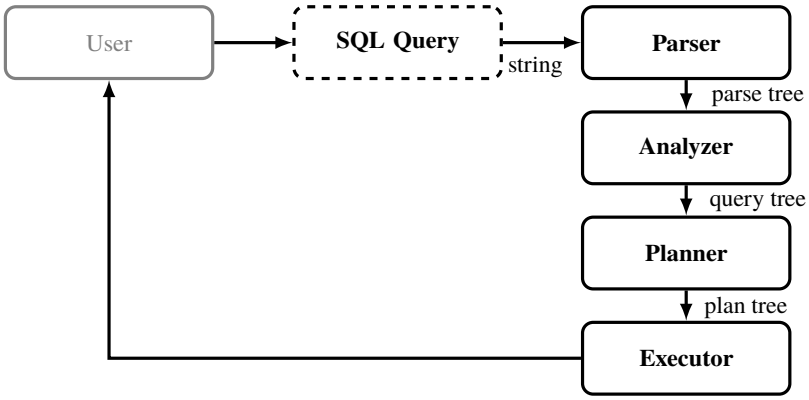

Fig. 1: PostgreSQL performs the same transformation steps for each received SQL query. Each module generates a new data structure ultimately resulting in a `plan tree`.

2.1 Plan Tree Printing and Parsing

The PostgreSQL-internal module defined in the file `outfuncs.c` can be used to *serialize* a plan tree into a *human readable* textual representation. The module defines the function `char* nodeToString(const Node* obj)` which takes a plan tree as input and generates the corresponding textual representation. Every node that can appear in a plan tree is associated with an output function.

In addition to the output function, every node must have an input function in the file `read.c`. The function `Node* stringToNode(char* str)` implements the *deserialization* of the textual representation and generates the corresponding internal node. The following applies: `stringToNode(nodeToString(node))` ≡ `node`.

However, the `stringToNode` function does not perform any *sanity checks*. This is problematic because the textual representation of a faulty plan will be parsed unnoticed as long as the syntax is correct. Normally, the `stringToNode` function is only used internally. Therefore, an error-free input is expected and no error messages are provided if the parsing fails.

The PostgreSQL executor relies on the planner to create error-free plans. There are several `Assert` statements implemented for each operator to support the development process, but they are disabled by default. Therefore, the executor is highly prone to errors in the plan tree structure.

It is possible to deserialize an arbitrary execution plan by using `stringToNode`, but it is extremely hard to develop or modify such a plan without assistance. A user would have to know and supply every field of any node in the correct ordering and format. It is very difficult to do this without making mistakes. Since we are about to assemble plans on our own, this places the burden on us to construct plans that are consistent and, in fact, executable at all. Below, we describe how to aid users in generating such correct plans.

```
{PLANNEDSTMT
    :commandType 1
    :parallelModeNeeded false
    :planTree
        {LIMIT
        :parallel_aware false
        :parallel_safe false
        :plan_node_id 0
        :targetlist [...]
        }
    :nParamExec 0
}
```

```
typedef struct PlannedStmt
{
  CmdType    commandType;
  ...
  bool   parallelModeNeeded;
  struct Plan *planTree;
  int    nParamExec;
} PlannedStmt;
```

Fig. 2: This shows a heavily truncated snippet of what the `nodeToString` function produces (left). On the right, we show how the corresponding data structure of a plan looks like.

2.2 Execution of Custom Plan Trees

PostgreSQL is designed to execute SQL queries. To our knowledge, there is no “side entrance” that allows the execution of plan trees directly. As mentioned, the stages before the executor are not required for the execution of a plan tree. PostgreSQL implements several methods to execute a query, using the SPI (Server Programming Interface) for example, but none of these functions bypass the steps prior to the executor.

Instead of writing a custom execution method from scratch, the extensive set of *hooks* provided by PostgreSQL can be used to inject a physical plan tree into the executor. Each hook consists of a global function pointer that allows to modify or replace the behavior of a

```
PlannedStmt *
planner(Query *parse, int cursorOptions, ParamListInfo boundParams)
{
     PlannedStmt *result;

     if (planner_hook)
         result = (*planner_hook) (parse, cursorOptions, boundParams);
     else
         result = standard_planner(parse, cursorOptions, boundParams);
     return result;
}
```

Listing 1: Entry point of the query optimizer [Gr18b, planner.c, 255–265]

module. Recompilation of core code is not required. A hook can be enabled and disabled by setting a global variable at run time of PostgreSQL.

We use the `planner_hook` to implement the execution of arbitrary plan trees. The `planner_hook` allows to replace the `standard_planner` with another function (see Listing 1). Plan forcing can be implemented by using a planner that ignores any input and, instead, return an execution plan that we have assembled on our own. Using the `planner_hook` we can avoid to deal with low-level error- and memory management, because the same execution methods as usual can be used.

The injection can now be done easily. After parsing an execution plan using the `stringToNode` function, SPI can be used to issue an arbitrary dummy query (e.g., `SELECT NULL`) while the `planner_hook` is enabled. PostgreSQL will parse, analyze and rewrite this query as usual. After the *analyze* module, the *query tree* is passed on to the planner. Usually, the `standard_planner` would now generate a plan tree for this query. But because the `planner_hook` is enabled, our custom planner is used instead. As explained earlier, this planner statically returns the previously parsed plan tree. This plan is evaluated by the executor accordingly. The injection is complete at this point. Figure 3 shows how the injection modifies the query pipeline.

3 Plan Forcing Applications

Now that we may lay “plan eggs into PostgreSQL’s nest”, a variety of interesting use cases open up. We sketch a few of these below.

3.1 Unnesting of Correlated Subqueries

Unnesting of *correlated subqueries* can greatly speed up query processing, but the majority of existing RDBMS cannot de-correlate queries in the general case. Thomas Neumann

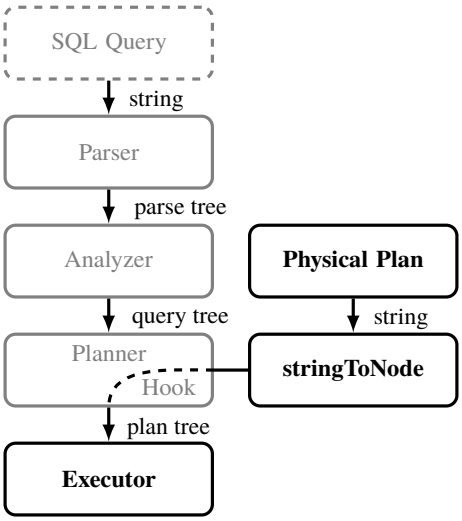

Fig. 3: Modified query pipeline that plan forcing implements. Modules shown in grey are active during normal SQL query processing but serve no function during plan forcing.

and Alfons Kemper provide a generic approach for unnesting arbitrary queries based on algebraic representations of a query with correlated subqueries [NK15]. As far as we know, this approach has not been implemented in any RDBMS besides HyPer [Ne11a].

The unnesting is performed using a rule system which implements algebraic rewriting of logical execution plans. The planner of the RDBMS HyPer uses their generic unnesting approach and is therefore able to unnest arbitrary queries whereas PostgreSQL in general is not. PostgreSQL's resulting performance disadvantage can be significant.

For PostgreSQL, there is no immediate way to use Neumann and Kemper's rule system. Ideally, the PostgreSQL developers would integrate this rule system directly with the optimization process of the planner. This would improve the performance of nested queries for all users. Because it is not clear if and when this planner feature will be implemented, we may instead use plan forcing to implement this. All that is left to do then, is the implementation of the rule system. This rewriting step can be implemented as a C language extension. In this case, the plan would stay inside the database system. Another option is to use the textual representation of a plan tree and rely on external tooling to implement the unnesting. Generally, such *externalization* of formerly strictly system-internal query processing steps is one of the most interesting aspects of the presented work.

Neumann and Kemper used two nested SQL queries Q_1 and Q_2 to compare the unnesting capabilities of various RDBMS, including PostgreSQL 9.1 and HyPer. They have been able to show that their rule system is able to produce unnested versions of these queries. The impact of their unnesting approach on PostgreSQL is not clear yet. However, instead of using

a SQL query as input, we can now use plan forcing to execute the unnested plans of Q_1 and Q_2 to determine the impact of unnesting for PostgreSQL. In effect, we use plan forcing to simulate a PostgreSQL kernel that is equipped with HyPer's sophisticated unnesting strategy. Among other lessons learned, this simulation can be used to decide whether far-reaching changes to PostgreSQL's current unnesting procedure are worth the effort.

Just as Neumann and Kemper have, we used $1,000$ `student` and $10,000$ `exam` tuples as test data. We then reproduced their experiment for Q_1 and Q_2 using PostgreSQL 10. The nested version of Q_1 had a run time of 3926.69 ms whereas the decorrelated formulation could be executed in just 47.77 ms. PostgreSQL 10 took $17,199.45$ ms to execute Q_2 without unnesting and $3,745.55$ ms with unnesting. The performance improvement matches those observed by Neumann and Kemper and therefore further supports the idea "that a system should be able to unnest arbitrary queries" [NK15].

3.2 Experimentation with Execution Plans

Plan forcing makes experimentation with execution plans possible because it can touch every aspect of a plan. This is interesting for research and educational purposes. Usually there is no way to locally control the behavior of the planner.

The plan tree data structure contains all information that the executor might need to evaluate a query. This makes these structures verbose and thus hard to construct from scratch. Normally, the user would not care about much of this information because the RDBMS automatically determines the required information based on the SQL query. We have to determine this information manually to construct a custom execution plan.

The complexity of the plan tree makes it extremely challenging to construct plan trees without assistance. We have developed a Haskell library that supports the creation of artificial execution plans. Based on two domain-specific-languages, we designed an *inference rule system* that uses the PostgreSQL catalog tables and the tree structure of the input to infer a multitude of gory plan details.

- PostgreSQL uses *object identifiers (OIDs)* as primary keys for various system tables. These OIDs are represented using an unsigned four-byte integer [Gr18a, 8.18 Object Identifier Types]. A plan tree consists of dozens of parameters that use OIDs to reference entries of the system tables. The inference rule system allows to use names such as "`sum`" or "`int4`" rather than the raw numeric value of the corresponding OID. This removes the necessity to memorize or lookup OIDs required for a plan. Also it adds a layer of abstraction to function and operator application, as they can be *overloaded*.
- Plan trees are a strongly typed data structure which means that every sub-expression must be annotated with a type. When provided with SQL queries, most of the required

typing is achieved by the *analyze* module. The sophisticated *type system* allows to skip explicit type annotations in SQL queries most of the time. This property is lost when an execution plan is constructed manually, however.
Using the same information from the system tables as the *analyze* module, our Haskell library implements a type system for the user input language. The type system serves multiple purposes in the inference rule system. First, it allows to infer type information almost completely. The only exception is the construction of constant values, which still requires explicit type annotations. Second, it is used to choose the correct alternative of possibly overloaded functions and operators, based on the number of arguments and their types.

These inferences ease plan construction notably, considering the fact that there are about 500 types, 800 operators, and 3000 functions in a fresh PostgreSQL 10.4 instance. Also semantical correctness can be guaranteed as it eliminates a potential source of error.

Another option we immediately get is to use plan trees as part of regular queries. Our PostgreSQL extension provides the table-valued SQL function `plan_execute` that expects an execution plan as input and returns the rows generated by this plan. Figure 4 shows how we can use this function to compose multiple plans into a single query. Queries that contain such rendered plans pass through the query pipeline as usual, but the *sub plan* encapsulated by the table-valued function is protected from any alterations by PostgreSQL.

```
SELECT a, b, a+b
FROM   plan_execute('{PLANNEDSTMT
                      :commandType 1 [...]
                      :planTree {SEQSCAN [...]}
                      :rtable ({RTE [...] :rtekind 0 :relid 90138 :relkind r [...])
                      [...]
                     }') AS tbl(a INT, b INT),
       plan_execute('{PLANNEDSTMT
                      :commandType 1 [...]
                      :planTree
                        {AGG [...]
                        :lefttree {SEQSCAN [...]}
                        :righttree <> [...]}
                      :rtable ({RTE [...] :rtekind 0 :relid 90138 :relkind r [...]})
                      [...]
                     }') AS agg(min INT)
WHERE a > min;
```

Fig. 4: Multiple occurrences of table-valued function `plan_execute` may be used to "stitch together" plan fragments as the users sees fit.

3.3 Plan Caching

We can turn a database system into its own plan cache, by storing the textual representation of execution plans in a table. This is an immediate result of this plan forcing approach. Instead of statically returning a plan tree, a different planner is required, that performs a lookup

in a plan cache table. If this lookup returns a result, we can omit the `standard_planner`. Otherwise, the caching planner has to call the `standard_planner` and add the generated plan into the cache table. Based on the mentioned Haskell library, we additionally have the option to prime the cache with execution plans that the original PostgreSQL query optimizer never would have considered on its own.

4 Related Work

Some RDBMSs such as Oracle, MariaDB and SQL Server natively support *optimizer hints* that can be used with SQL statements to alter execution plans. "Hints let you make decisions usually made by the optimizer. Hints provide a mechanism to instruct the optimizer to choose a certain query execution plan based on the specific criteria. For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to choose a more efficient plan than the optimizer" [Or18, 16 Using Optimizer Hints].

Plan forcing is comparable to optimizer hints, but their expressive strength is noticeably weaker. While optimizer hints, for instance, allow to choose a certain join algorithm locally, they can not be used to create artificial plan trees. The planner and a SQL query is still required to generate a certain plan.

"Appropriate use of the right hint on the right query can improve query performance. The exact same hint used on another query can create more problems than it solves, radically slowing your query and leading to serve blocking and timeouts in your application [Ne11b, p. 177]. While query hints allow you to control the behavior of the optimizer, it doesn't mean your choices are necessarily better than the optimizer's choices. [. . .] Also, remember that a hint applied today may work well but, over time, as data and statistics shift, the hint may no longer work as expected [Ne11b, p. 181]."

SQL Server 2005 introduced the `USE PLAN` query hint. This allows to specify an entire execution plan as a target to be used to optimize a query. The `USE PLAN` hint can force most of the specified plan properties, including the tree structure, join order, join algorithms, aggregations, sorting, unions, and index operations like scans [Ne11b, pp. 248–250]. This approach is similar to ours, as it allows the execution of a serialized *plan tree*. However, plan forcing in SQL Server still requires a SQL query and it is required that the *plan tree* can be produced by the optimizer's normal search strategy [Pa05]. These restrictions prevent SQL Server from crashing when an invalid *plan tree* is encountered. This is not the case for PostgreSQL. Invalid *plan trees* can crash the database server which results in a restart of the service.

The `pg_plan_advsr` extension is another interesting use case of the `planner_hook`. This extension implements a feedback loop from the executor to the planner which is used to auto-tune plans. Normally, no run-time information of plans is fed back into the planner in

order to self optimize. By repeatedly feeding back the information about the actual execution of a plan, the error in estimation of row counts can be corrected. That affects the planning process and the resulting plan [Ya18].

5 Closing Thoughts

We were able to add plan forcing to PostgreSQL by using a C language extension and standard features such as the `planner_hook`. This demonstrates impressively how far the envelop of PostgreSQL extensibility can be pushed. Further investigations into the various application opportunities of plan forcing will be required. An important module we are currently working on is another Haskell library that abstracts physical into logical algebra plans. One application of this library will be algebraic rewriting. More specifically, we want to use it to implement Neumann and Kemper's unnesting rule system for PostgreSQL, as described in Section 3.2.

References

[Ch98] Chaudhuri, S.: An Overview of Query Optimization in Relational Systems. In: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. PODS '98, ACM, Seattle, Washington, USA, pp. 34–43, 1998, ISBN: 0-89791-996-3, URL: http://doi.acm.org/10.1145/275487.275492.

[Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks -. Freuburg i.B., 1970.

[DD03] Douglas, K.; Douglas, S.: PostgreSQL - A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases. Sams Publishing, Indianapolis, Indiana, 2003.

[Gr18a] Group, T. P. G. D.: PostgreSQL 10 Documentation, 2018, URL: http://www.postgresql.org/docs/10/static/index.html, visited on: 06/25/2018.

[Gr18b] Group, T. P. G. D.: PostgreSQL 10 Source Code, 2018, URL: https://www.postgresql.org/ftp/source/v10.0/, visited on: 06/25/2018.

[Le15] Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T.: How Good Are Query Optimizers, Really? Proc. VLDB Endow. 9/3, pp. 204–215, Nov. 2015, ISSN: 2150-8097, URL: http://dx.doi.org/10.14778/2850583.2850594.

[Ne11a] Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4/9, pp. 539–550, June 2011, ISSN: 2150-8097, URL: http://dx.doi.org/10.14778/2002938.2002940.

[Ne11b] Nevarez, B.: Inside the SQL Server Query Optimizer -. Red Gate Books, 2011.

[NK15] Neumann, T.; Kemper, A.: Unnesting Arbitrary Queries. In: BTW. Vol. 241. LNI, GI, pp. 383–402, 2015.

[Or18] Oracle: Oracle Database Online Documentation, 2018, URL: https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327, visited on: 06/25/2018.

[Pa05] Patel, B. A.: Forcing Query Plans, 2005, URL: https://technet.microsoft.com/en-us/library/cc917694.aspx, visited on: 06/25/2018.

[Ya18] Yamada, T.: Auto Plan Tuning using Feedback Loop, 2018, URL: https://www.postgresql.eu/events/pgconfeu2018/schedule/session/2132-auto-plan-tuning-using-feedback-loop/, visited on: 10/31/2018.