Devel-Op: An Optimizer Development Environment

Zhibo Peng, Mitch Cherniack and Olga Papaemmanouil

Computer Science Department, Brandeis University, Waltham, MA
{docp, mfc, olga}@brandeis.edu

***Abstract*—Recent advances in the underlying architectures of database management systems (DBMS) have motivated the redesign of key DBMS components such as the query optimizer. Optimizers are inherently difficult to build and maintain, and yet there exists no software engineering tools to facilitate their development. In this paper, we introduce a [Devel]opment Environment for Query [Op]timizers (Devel-Op) designed to facilitate the *rapid prototyping*, *profiling* and *benchmarking* of optimizers. Our current version of the tool permits declarative specification and generation of two key optimizer components (the logical plan enumerator and physical plan generator) as well as debugging and visualization tools for profiling generated components.**

I. Introduction

In recent years, new data management systems have emerged that exploit new hardware technologies (e.g., flash memory [1]) to outperform the traditional DBMS. As a result, the designs of many key DBMS components are being revisited in light of new assumptions. For example, new approaches to query optimization have been proposed that assume a distributed shared-nothing architecture rather than a centralized client-server architecture [2], flash memory storage rather than disks [3], column-stores rather than row-stores [4] and compressed data rather than uncompressed [5].

New approaches to query optimization are hindered by the absence of software engineering tools that support the *rapid prototyping*, *profiling* and *benchmarking* of optimizers and their components. In this paper, we introduce **Devel-Op**: a [Devel]opment Environment for Query [O]ptimizers consisting of such tools. Our current version of Devel-Op includes support for two key components of the query optimizer: the *Logical Plan Enumerator* (*LPE*) and the *Physical Plan Generator* (*PPG*). Rapid prototyping support is provided by way of declarative specification languages (*LSL* and *PSL*) and corresponding LPE and PPG component generators. Profiling support is provided with *debugger* and *visualization* tools for tracing the operation of these components over any query.

II. Background

Figure 1 shows a generic query optimizer architecture assumed by **Devel-Op**. A *Logical Plan Enumerator* (LPE) accepts a representation of a given query such as a *join graph*, and enumerates a set of logical plans that constrain the physical plans that the optimizer should consider. These logical plans are then sent to a *Physical Plan Generator* (PPG) which produces a set of physical plans for each logical plan by replacing each logical join with a physical join operation (e.g., merge join), determining which input to that join is outer and which is inner, and replacing each reference to a table with a physical access method (e.g., file scan). The *Plan Pruner* then selects subsets of these plans on the basis of their estimated costs or other "interesting" properties.

The operations of the LPE, PPG and Plan Pruner are typically threaded. For example, bottom-up plan generation (as in System R [6]) interrupts an LPE after it has produced some set of logical subplans, invokes the PPG to produce a set of corresponding physical subplans and then invokes the Plan Pruner to prune this set. The LPE then resumes, but working only with those logical plans that survive pruning. The role of the *Controller* is to schedule the LPE, PPG and Plan Pruner threads (i.e., deciding when to transfer control among them) and thereby determines whether pruning is performed eagerly on subplans (bottom-up plan generation) or only once entire plans are constructed (top-down plan generation).

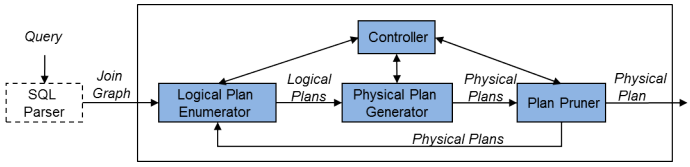

Fig. 1. A Generic Query Optimizer Architecture

Figures 2b-d show examples of the query representations passed between optimizer components for the TPC-H based query shown in Figure 2a.[1] The join graph for this query, consisting of nodes for each table and edges between those for which the query has join predicates, is shown in Figure 2b. Figure 2c shows a logical plan (annotated with logical *plan properties*, `OrdSet` and `Rank`) that could be enumerated for this query by the LPE. Figure 2d shows a physical plan that could be constructed from this logical plan by the PPG. The encapsulation of common query optimizer behaviors with these dedicated components make it straightforward to prototype and evaluate different approaches to their design.

III. Devel-Op

Our current version of **Devel-Op** [7] supports component-specific prototyping and profiling. Therefore it offers a systematic solution for the development of well-modularized query optimizer that clearly distinguishes us from existing work on profiling optimizers (e.g., Picasso Visualizer [8]) and providing extensibility in query optimization (e.g., EvitaRaced [9]). In

[1]Tables `L`, `O`, `C` and `N` are short for TPC-H tables, `Lineitem`, `Orders`, `Customer` and `Nation` respectively.

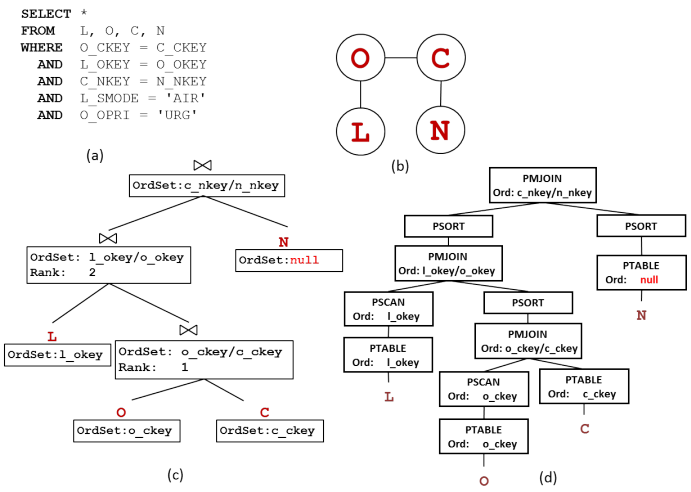

Fig. 2. A query (a), its join graph (b), and logical (c) and physical (d) plans

```
1  plan ::  tablePlan | selectPlan | joinPlan;
2
3  tablePlan :: TABLE '(' T=ID ')'
4     $OrdSet = ALL (Ord);
5  selectPlan :: SELECT '(' A=ID '=' g=LITERAL, L=plan ')'
6     $OrdSet = ALL (Ord);
7  joinPlan :: JOIN '(' A1=ID '=' A2=ID, L1=plan, L2=plan ')'
8     $OrdSet = ALL (Ord);
9     $Rank =  IF ($A1.val \in $L1.OrdSet) AND
10                $A2.val \in $L2.OrdSet) THEN 1
11            ELSIF ($A1.val \in $L1.OrdSet) OR
12                $A2.val \in $L2.OrdSet) THEN 2 ELSE 3
```

Fig. 3. An LPE Specification in LPL

the following paragraphs we describe in detail our unique component specification and generation process as well as our profiling and visualization tool.

A. Component Specification and Generation

We have defined declarative specification languages *LSL* and *PSL* for specifying the LPE and PPG optimizer components respectively, and component generators that generate LPE and PPG components from their corresponding specifications. Specifications in LSL resemble *attribute grammars* as found in compiler generation tools such as Yacc [10], OX [11] and Antlr [12]. The grammar of an LSL specification defines the logical plan tree that is enumerated by the generated LPE, while attributes (denoted with '$') specify the logical *plan properties* (e.g., result cardinality) associated with each node in the tree. Note that the specification shown in Figure 3 is very simple, assuming logical plans consisting solely of tables (`tablePlan`), simple selections (`selectPlan`) and equijoins (`joinPlan`). The simplicity of this example facilitates presentation and in no way reflects limitations on the expressivity of LSL specifications.

The LSL specification shown in Figure 3 defines two logical plan properties: `OrdSet` and `Rank`. `OrdSet` denotes the set of all sort orders (columns) that hold of any physical plan that could be generated from a given logical plan. For example, the subplan denoting $\texttt{L} \bowtie \texttt{O}$ in Figure 2c has `OrdSet` equal to {`o_okey / l_okey`}, indicating that there is just one sort order (on column `o_okey`, which because of the join predicate on `L` and `O`, has alias `l_okey`) resulting from the physical plans generated from this logical plan. As we will see below, this is because the simplistic physical plan generator specified in Figure 4 only generates a merge join from logical joins. If, for example, it also generated an indexed nested loop join with `O` as the outer relation, then `OrdSet` would also contain the sort orders associated with table `O`. The LSL expression, `ALL (att)` specifies that the value of this property is the set of all values for the physical property named *att* in all physical plans generated from a given logical plan.

All LSL join specifications must define the special property, `Rank` which is used by generated LPEs to determine which join orderings to enumerate for any given query. A join rank should be high (i.e., close to or equal to 1) if it is a join that should be performed early in a plan such as a *local join* requiring no data transfer between nodes prior to performing the join in a distributed DBMS), and a higher rank otherwise. The LPE enumeration algorithm then converts join graphs into logical plans incrementally by converting each edge and the nodes it connects, in rank order, into a *join node* denoting a logical join subplan. Thus, generated LPEs enumerate only those logical plans whose join subplans have rank equal to or higher than any join plans that contain them. For the LPE specification of Figure 3, `Rank` assigns a join a rank of 1 if both of its inputs are potentially ordered on their respective join attributes (meaning that the join could be implemented as a merge join without requiring prior sorting of its inputs), 2 if one of the inputs is so ordered and 3 otherwise.

```
1   L: tablePlan -> P: ptablePlan ::
2   TABLE ($T)
3     -> R=PTABLE ($T)
4        $R.Ord = LK.ord($T.val)
5
6   L: selectPlan -> P: pselectPlan ::
7   SELECT ($A '=' $K, $L)
8     -> R=PSCAN ($A '=' $K, $P1) WHEN($L -> $P1)
9        $R.Ord = $P1.Ord
10
11  L: joinPlan -> P: pjoinPlan ::
12  JOIN ($A1 '=' $A2, $L1, $L2)
13    -> R=PMJOIN ($A1 '=' $A2, $P1, $P2)
14       WHEN ($L1 -> $P1, $L2 -> $P2,
15             $A1.val = $P1.Ord, $A2.val = $P2.Ord)
16       $R.Ord = [$A2.val/$A1.val]
17    ->  R=PMJOIN ($A1 '=' $A2, PSORT($A1, $P1),$P2)
18       WHEN ($L1 -> $P1, $L2 -> $P2,
19             !($A1.val = $P1.Ord), $A2.val = $P2.Ord)
20       $R.Ord = [$A2.val/$A1.val]
21    ->  R=PMJOIN ($A1 '=' $A2, $P1, PSORT($A2,$P2))
22       WHEN ($L1 -> $P1, $L2 -> $P2,
23             $A1.val = $P1.Ord, !($A2.val = $P2.Ord))
24       $R.Ord = [$A2.val/$A1.val]
```

Fig. 4. A PPG Specification in PSL

PPGs are specified in specification language PSL as shown in Figure 4. PSL specifications resemble *Tree Attribute Grammars (TAGs)* [13] that are commonly used in compiler generation to specify language translators. TAGs differ from standard attribute grammars in that they consist of *translation* rather than *grammar* rules with left-hand sides that are parse tree expressions (from the language being translated) rather than

nonterminals. Analagously with LSL, the attributes of PSL denote *physical* plan properties which annotate the nodes of the physical plan trees that generate PPGs construct.

The PPG generated from the PSL specification of Fugure 4 constructs the physical plan shown in Figure 2d (as well as other physical plans) from the logical plan of Figure 2c. This specification indicates how to translate each logical subplan into its equivalent physical plans; for this example, a file scan (`pscan`) for the logical select plan, a merge join (`pmjoin`) for the logical join plan, and a table reference (`ptable`) for the logical table plan. Observe that there are three translation rules (each denoted by "`->`") for translating a logical join into physical merge joins with each rule predicated by the conditions delimited by the reserved word, `WHEN`. The rule of line 13 says that a logical join can be translated into a physical merge join without prior sorting if both inputs are already sorted on their respective join attributes. The rules of lines 17 and 21 say that if just one of the join inputs is sorted, a merge join must be preceded by a sort of the other join input. All physical plans generated according to this specification includes the physical plan property, `Ord`, which specifies the attribute on which a plan's result will be sorted (or is *null* if the result is not sorted). This property is used in two ways: to determine if an input to a merge join needs to be sorted, and in determining the set of values associated with the logical property, `OrdSet`.

PSL includes two unique features not found in traditional TAGs:

Multiple mappings: Unlike traditional translators which map every expression of one language into a single expression of another language, physical plan generation might map a given logical plan into *several* physical plans with selection occurring later in the optimization process (i.e., during plan pruning). Thus, PSL specifications allow each logical plan expression to be associated with multiple translation rules, with each rule applied only when some set of conditions (expressed in PSL with a `WHEN` clause) is satisfied. Space limitations prevent us from showing other join translation rules such as hash join or indexed nested loop join for the example of Figure 4, but such rules could lead a logical join plan to be translated into several different physical join plans.

Shared properties: As we saw with the `OrdSet` property in the LSL specification of Figure 3, physical plan properties such as `Ord` can be shared with logical plan specifications as in the definition of `OrdSet` which was defined as the set of all values for `Ord` taken from all physical plans resulting from translation of a logical plan. Property sharing works in the other direction also; logical plan properties are implicitly shared with the physical plans into which they are translated.[2]

B. Debugging and Visualization

Figure 5 shows a screenshot from the **Devel-Op** environment assuming a LPE and PPG generated from the examples

[2]Of course, problems can arise from the cycles of shared properties, but our component generator tools detect and forbid such cycles.

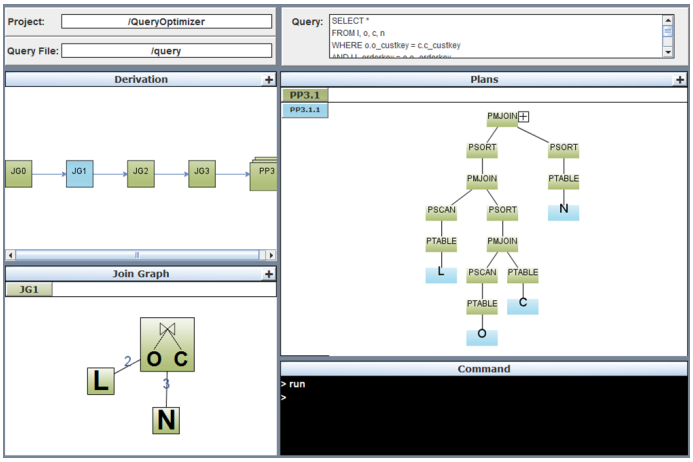

Fig. 5. A Screenshot of the **Devel-Op** Debugger and Visualizer

of Figures 3 and 4, and the query of Figure 2a. The *Derivation* window gives an overview of the optimization process on the current query. Nodes in this window denote join graphs, logical plans, or sets of physical plans. By selecting different derivation nodes, users can view the corresponding state of optimization consisting of: the current join graph (shown in the *Join Graph* window), as well as the set of logical or physical plans produced at that step (shown in the *Plans* window). Users can also examine the values of properties of join graph or plan nodes. The *Command* window is where a user can control the optimization process using standard commands as found in debuggers for programming languages such as GDB for Gnu C++ [14]. Debugging commands include:

1) `next`: used to join the highest ranked edges of current selected join graph to generate the intermediate join graphs, or generate the physical plans from current selected logical plan
2) `run`: used to generate all logical and physical plans generated from the join graph or logical plan currently selected in the *Join Graph* window, and
3) `complete`: used to generate all remaining logical and physical plans from the entire query.

IV. A Demonstration Scenario

We will demonstrate the version of **Devel-Op** available as of the time of the conference. Our current version includes declarative specification languages for the LPE and PPG components of the optimizer and component generators that generate LPE and PPG components from their specifications and debugging and visualization tools that enable profiling of generated optimizers. We also expect to include generation tools for Plan Pruners and possibly Controllers. In the absence of specifiable Plan Pruners and Controllers, generated optimizers will use a default bottom-up Controller and prune the set of generated physical plans at the end to single physical plan using a simple built-in I/O based cost model as in [15].

Suppose that a developer for some DBMS vendor specifies an optimizer based on the LPE specification of Figure 3,

and the PPG specification of Figure 4. After compiling these specifications using our generators, the developer might profile the resulting optimizer using TPC-H queries such as that of Figure 2a, and see the result of optimization exhibited in the **Devel-Op** environment as shown in Figure 5. Looking at the result of optimization, the developer might notice that the optimizer did not select the desired plan for this query (shown in Figure 7b). He then needs to determine which component of the optimizer is at fault; the LPE if it failed to enumerate the proper logical plan (the one shown in Figure 7a), the PPG if the right logical plan was enumerated but not translated into the desired physical plan, or the pruner if the desired physical plan was constructed but not selected. The developer would need to examine the set of logical and physical plans generated to see where correction is required.

Suppose that examination of enumerated logical plans reveals that the desired logical plan shown in Figure 7a was never enumerated by the LPE. To see why this was the case, the developer might trace the operation of the LPE as illustrated in Figure 6. Figure 6a shows the initial join graph (JG0) for this query with edges annotated with their ranks. The developer might issue the command `next` to see the join graph of JG1 shown in Figure 6b, and see that `L` will be joined with `O` ⋈ `C` before `N` is in all enumerated logical plans. Thus, the fault lies in the ranking for the join of `O` ⋈ `C` with `N`; the rank should be higher because one of the inputs to this join (`N`) is small. The developer would then modify the LPE specification, modifying the formula determining `Rank` to also consider another logical plan property (`Size`) returning a cardinality estimate for each plan. After modifying and recompiling this specification, a trace of the resulting optimizer over the same query would be as shown in Figure 6c and d, leading to enumeration of the desired logical plan shown in Figure 7a. At this point, if the desired physical plan is still not selected, the developer would need to see if the PPG constructed this plan for the optimizer to consider. If it didn't, the PPG specification would need to be modified as well. If it did, then the problem would lie with the Pruner and its associated cost model.

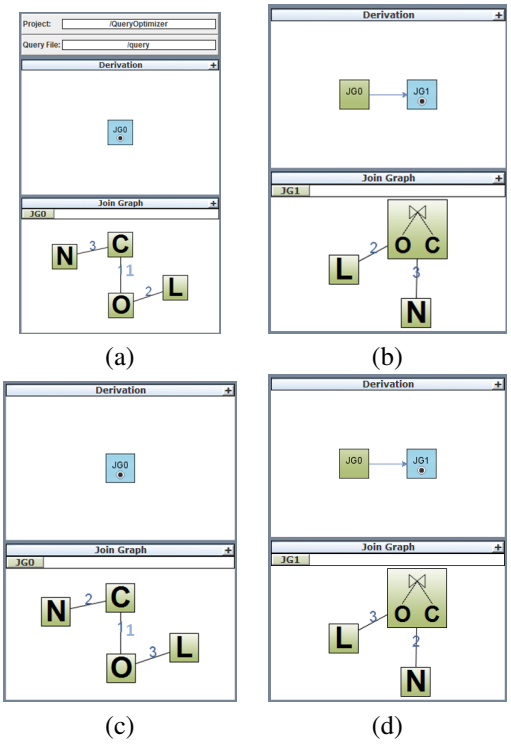

Fig. 6. A Debugging Trace with **Devel-Op**

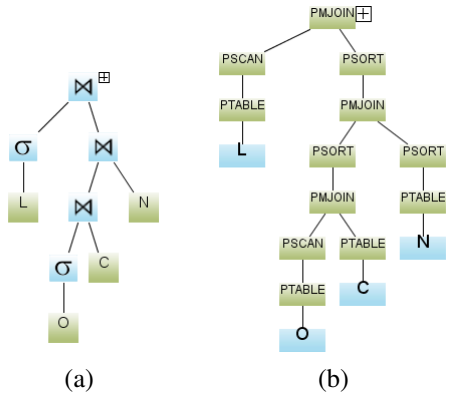

Fig. 7. Desired Logical (a) and Physical (b) Plans for Query of Figure 2a

V. Conclusion

This demonstration shows that our novel query optimizer development environment (**Devel-Op**) addresses the rapid prototyping and profiling needs of query optimizer developers and researchers.

References

[1] P. Bonnet, L. Bouganim, I. Koltsidas, and S. Viglas, "System co-design and data management for flash devices," *PVLDB*, pp. 1504–1505, 2011.

[2] S. Krishnamoorthy, A. K. Saple, and P. H. Achutharao, "An integrated query optimization system for data grids," in *COMPUTE '08*, 2008.

[3] Tsirogiannis et al, "Query processing techniques for solid state drives," in *SIGMOD*, 2009.

[4] Stonebraker et al, "C-store: a column-oriented dbms," in *VLDB*, 2005.

[5] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems," in *SIGMOD*, 2001.

[6] Selinger et al, "Access path selection in a relational database management system," in *SIGMOD*, 1979.

[7] "Devel-Op Project Website, http://www.cs.brandeis.edu/ develop."

[8] J. R. Haritsa, "The Picasso Database Query Optimizer Visualizer," in *VLDB*, 2010.

[9] Condie et al, "EvitaRaced: Metacompilation for Declarative Networks," in *VLDB*, 2008.

[10] S. C. Johnson, "Yacc: Yet another compiler-compiler," 1975.

[11] C. Fl, K. M. Bischoff, and K. M. Bischoff, "Ox: An attribute-grammar compiling system based on yacc, lex, and c," 1993.

[12] T. J. Parr, T. J. Parr, and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," 1995.

[13] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[14] "GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/."

[15] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*, 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006.