A Development Environment for Query Optimizers

Olga Papaemmanouil, Nga Tran, Mitch Cherniack
Brandeis University
{olga, nga, mfc}@cs.brandeis.edu

ABSTRACT

New research has emerged that revisits the design of various components of a data management system to determine how their designs can exploit the targeted applications and underlying architecture of that system. To support this effort, we introduce a development environment that supports the rapid prototyping, evaluation and refinement of query optimizer components for targeted data management systems. The key features of this development environment are component generator tools that process declarative specifications of the components, component-specific benchmarks and profiling tools. We describe some initial results specific to one optimizer component (the join plan enumerator) and we present some preliminary thoughts about the remaining environment design.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools, Tracing*

General Terms

Design, Experimentation

1. INTRODUCTION

In recent years, new *targeted data management systems* have emerged that exploit modern architectures to outperform the DBMS 'elephants' in application areas that lie beyond the traditional sweet spot of business data processing (e.g., stream processing and data warehouse engines). This trend demonstrates our long-held contention that "one-size-does-not-fit-all" in data management systems [8], and it naturally follows that "one-design-does-not-fit-all' for the components of these systems. For example, recent work has shown how the design of a query optimizer can benefit from consideration of properties that distinguish the underlying data management system (e.g., distributed shared-nothing architecture rather than centralized [5], flash memory storage rather than disks [9], and column-based data layout rather than row-based [1]). As the trend towards targeted data management systems continues, we expect to see competing query optimizer designs proposed that leverage the properties of those systems. To support these efforts, we are designing a *development environment* that facilitates the engineering of system-specific optimizer component designs.

DBTest'10, June 7, 2010, Indianapolis, Indiana, USA.

In this paper, we present an early-stage development environment for query optimizer components that are collectively responsible for *join plan selection*: the selection, for any input query, of a physical execution plan which specifies 1) a join execution order, 2) physical join operators and 3) outer and inner inputs for each join in that query. The components include: a *Join Plan Enumerator* (*JPE*) that enumerates a set of *logical join plans* (essentially, join orders) that constrain the physical plans an optimizer will consider, a *Physical Plan Mapper* (*PPM*) that maps each logical join plan to a set of physical plans, a *Pruner* that restricts join plans on the basis of their estimated costs or other "interesting" properties, and an optimizer *Controller* that controls the scheduling of the JPE, PPM and Pruner. The key features of the development environment are:

1. *Rapid Prototyping*, which facilitates the generation and refinement of multiple designs of a given component. To support the rapid prototyping of optimizer components, our proposed development environment includes *generator tools* that automatically generate components according to provided *declarative specifications*.
2. *Component Evaluation*, which makes it possible to compare competing designs for a given optimizer component. To support component evaluation, our development environment follows the existing trends of isolating the behavior of components [2] and hence includes component-specific *benchmarks* that measure the "goodness" of an optimizer component design independently of other components. Such benchmarks makes it possible, for example, to compare different join plan enumeration strategies independently of any cost model that might be used to select from enumerated plans, and thereby show how *robust* an enumeration strategy might be in the presence of errors or evolving versions of a cost model.
3. *Component Refinement*, which allows a designer to easily modify and thereby improve a component's design. To support component refinement, our proposed development environment includes component profiling tools (such as statistics-gathering and visualization tools) that help isolate flaws in a component design. Once flaws are detected, the specification for the component can be modified, the component's generator tool is used to generate the version of the component and the component benchmarks can be used to evaluate the modified component.

The remaining of this paper is organized as follows. In Section 2 we describe a generic architecture for join plan selection and provide an overview of our development environment. In Section 3 we describe the design and implementation of the development for one of these components (the JPE) and describe how the experience of generating different JPE components provided insight into JPE benchmarks and profiling tools. In Section 4 we sketch our current thoughts on how to approach the development for the remaining components and we conclude with final remarks in Section 5.

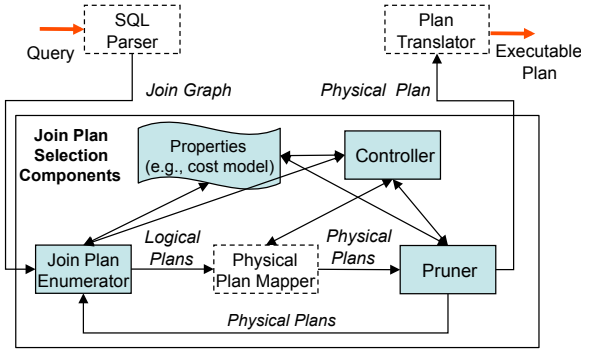


Figure 1: Join Plan Selection Architecture

2. ENVIRONMENT OVERVIEW

Figure 1 shows a generic architecture for the components of an optimizer that perform join plan selection: the *Join Plan Enumerator (JPE)*, the *Physical Plan Mapper (PPM)*, the *Pruner* and the *Controller*. A JPE begins the join plan selection for a query by processing the query *join graph*: an undirected graph, $G = (V, E)$, consisting of a set V of tables named in the query (annotated with any selection predicates on those tables), and a set E of edges denoting pairs of tables joined by the query. The JPE produces a *candidate plan set* consisting of one or more (logical) *join plans*, each of which specifies an ordering of the query joins. Thus, the candidate plan set constrains the *physical plans* that can be chosen by the optimizer.

A candidate plan set is handed off to a PPM that maps each join plan to a set of physical plans. A physical plan is derived from a join plan by replacing each logical join with a physical join operation (e.g., hash-join), and each reference to a table with a physical access method (e.g., index-scan). Sets of physical plans derived by the PPM are then passed to a Pruner which prunes plans according to a *cost model* and potentially, additional properties of query plans (e.g., *interesting orders*).

The operations of the JPE, PPM and Pruner are typically threaded. For example, bottom-up plan generation (as in System R [7]) will interrupt a JPE after it has produced some set of *partial* join plans, invoke the PPM to produce a set of corresponding partial physical plans and then invoke the Pruner to prune this set. The JPE then resumes, but working only with those join plans that survived pruning. The role of the Controller is to schedule the JPE, PPM and Pruner threads (i.e., deciding when to transfer control from the JPE to the PPM and Pruner) and thereby determines whether pruning is performed eagerly on subplans (as in bottom-up plan generation) or only once entire plans are constructed (as in top-down plan generation).

The architecture shown in Figure 1 is conceptual – the design of any given optimizer might be modularized differently than shown here. But the 3-step approach to join plan selection is universal, and the encapsulation of each step within a dedicated component makes it straightforward to prototype, evaluate and refine a variety of component designs.

Our development environment supports the design of components by including for each: 1) a *generator* tool that uses a component's *declarative specification* to generate executable component code to be 'plugged' into the framework of Figure 1, thereby supporting component prototyping, 2) a set of component-specific *benchmarks* that enable the comparisons of competing component designs independent of the designs of other components, thereby supporting component evaluation, and 3) component *profiling tools* (e.g., for *statistics gathering* and *visualization*) that help to expose design flaws that lead a component to behave in undesirable ways, thereby supporting component refinement. Our development environment also includes a *generator* and an associated *specification language* for the (logical and physical) plans that are processed by these components and for the inferable plan properties (*cost*, *orders* etc.) that are available for use by any component.

3. EARLY WORK: DEVELOPING JPE'S

The role of a JPE is to construct, for any query, a *candidate plan set* consisting of logical join plans for an optimizer to consider. The key issue in JPE design is constraining the size of the candidate plan set. Early optimizers such as System R [7] imposed a structural constraint on plans, considering only those whose joins have at least one base relation as input (*left-deep* enumeration). However, for many queries, the best query plan is bushy rather than left-deep [6]. Another proposed approach is *exhaustive* enumeration of all possible plans. But this approach can also lead to the selection of a poor plan since it excludes no plan and therefore cannot not scale to complex queries but also due to the inherent brittleness of cost models that can lead an optimizer to incorrectly predict the cheapest plan [2]. Thus, most DBMSs provide an *optimization level* "knob" which determines when the system will interrupt enumeration, but which may also prevent an optimal plan from ever being considered.

Rather than assuming a 'universal' (e.g., left-deep or exhaustive) approach to join plan enumeration, our development environment supports the construction of *targeted JPEs* that consider semantic and architecture-specific physical properties of the queried data in choosing which plans to enumerate. For example, a targeted JPE for a distributed, shared-nothing DBMS might enumerate only and all plans where *local joins* (i.e., joins that do not require any prior data transfer) are executed before *non-local joins*. If designed well, a targeted JPE will produce small candidate plan sets that contain good plans and avoid poor ones. Compared to an exhaustive enumerator, a well-designed targeted JPE is more scalable and more resilient in the presence of an imperfect cost model given that an optimizer can only choose as poor a plan as it is constrained to consider.

3.1 Specification and Generation

We have designed and constructed development tools to assist in the engineering of targeted JPEs. In this section, we describe our JPE *generator tool* that generates JPEs according to input *specifications*. Our approach to specifying JPEs is based on the intuition that good join plans tend to be those that perform certain types of joins (e.g., local joins) early in a plan and other types later. Thus, we specify a JPE with a set of *ranking functions*, each of which maps any join appearing in a query to its relative rank (with ties allowed). For any ranking function r and query Q, the candidate plan set of Q *induced by* r consists of all plans that satisfy the constraint that inputs to every join in the plan are either base relations or joins of higher rank according to r. When applied to a query, Q, the JPE generated according to the set of ranking functions, R, returns the union of candidate plan sets induced by the ranking functions contained in R.

Algorithm `JPEG`**.** Figure 3 shows the algorithm (`JPEG`) used by JPEs generated by our generator tool to construct the candidate set of plans for a query Q induced by one of the ranking functions $r \in R$. A query Q is represented by the join graph, $G = (V, E)$ where V is a set of nodes representing the tables referenced in Q and E is a set of edges connecting tables that are joined in Q. `JPEG` incrementally converts each edge in E

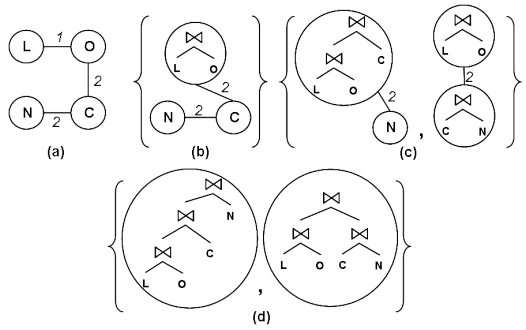


Figure 2: Tracing the Construction of a Candidate Plan Set

```
ALGORITHM JPEG (r, V_q, E_q)
 Res := {(V_q, E_q)}
 WHILE ∃(V,E) ∈ Res where |E| > 0 DO
   E_max := {(v1,v2) ∈ E | r(v1,v2) is maximal in E}
   JGs := ∅
   FOR EACH (v1,v2) ∈ E_max DO
     V' := V − {v1,v2} ∪ {v1 ⋈ v2}
     E⁻ := {(u,w) ∈ E | w ∈ {v1,v2}}
     E⁺ := {(u,j) | (u,w) ∈ e⁻}
     E' := E − E⁻ ∪ E⁺
     JGs := JGs ∪ {(V',E')}
   Res := Res − {(V,E)} ∪ JGs
 RETURN Res
END
```

Figure 3: The JPEG generation algorithm

and its associated endnodes to a single *join node* that represents a plan that joins the tables connected by the converted edge. Edges are converted in rank order from highest to lowest according to r. Thus, in each step the number of nodes (and edges) in G is reduced by one until there is just 1 node remaining.

This is illustrated in Figure 2. Figure 2a shows a join graph for a query that joins 4 tables (L, O, C and N), whose edges have been annotated with their rank according to some ranking function. Figure 2b shows the join graphs that results from converting the highest-ranked edge in the join graph of Figure 2a ((L, O)) to a single join node (L ⋈ O). Note that the edge connecting C to O in Figure 2a is modified to connect C to this join node in Figure 2b. Because the join graph of Figure 2b has 2 maximally ranked edges, the result of processing this join graph (Figure 2c) is a set containing two join graphs: each resulting from converting one of these edges. Finally, Figure 2d shows the set resulting from converting the remaining edge in each of the join graphs in Figure 2c. This set, containing join graphs which only consist of a single node, is the candidate plan set produced by the JPEG algorithm, with each node denoting a plan in that set.

A JPE for a Centralized, Client-Server DBMS. We used our generator tool to generate a JPE ($\texttt{JPE}_\texttt{C}$) for the centralized, client-server DBMS, SQL Server using a set, $\texttt{R}_{\texttt{JPE}_\texttt{C}}$, consisting of 3 ranking functions. These ranking functions categorize every join, J, appearing in a query according to the following properties: (1) the *[C]ardinality* of J ($n :: 1$ vs $n :: m$)), (2) the *[S]ize* of each of J's input relations (e.g., both inputs are *small* vs one of J's inputs is *large*), (3) the *Sort [O]rder* of each of J's input relations (e.g., *both* inputs are sorted on their respective join attributes vs *neither* are), and (4) the *[I]ndexes* defined on each of J's input relations (e.g., *both* inputs have indexes on their respective join attributes vs *neither* do). Each ranking function in $\texttt{R}_{\texttt{JPE}_\texttt{C}}$ ranks the "values" of each property equivalently (e.g., for Cardinality, $n :: 1$ is higher-ranked than $n :: m$), but differs by the relative "weight" attached to each property (e.g., one ranking function in $\texttt{R}_{\texttt{JPE}_\texttt{C}}$ ranks first on Cardinality, then Size, then Sort Order, and then Indexing, while another swaps the relative importance of Size and Sort Order.

Query (# Tables)	SS		SS + $\texttt{JPE}_\texttt{C}$	
	Selected Plan Runtime (sec)	Distinct JWUs	Selected Plan Runtime (sec)	Distinct JWUs
Q2 (5)	13.27	26	13.27	15
Q3 (3)	267.49	4	267.49	4
Q5 (6)	332.94	244	254.28	77
Q7 (6)	883.74	137	205.62	57
Q8 (8)	304.08	507	304.08	204
Q9 (6)	457.52	255	395.88	76
Q10 (4)	329.62	11	329.62	10
Q11 (3)	40.70	4	40.70	4
Q18 (3)	1392.86	4	1392.86	4
Q21 (4)	259.47	11	259.47	8

Table 1: SQL Server (SS) vs SQL Server + $\texttt{JPE}_\texttt{C}$ (SS + $\texttt{JPE}_\texttt{C}$)

To evaluate $\texttt{JPE}_\texttt{C}$, we used it to generate candidate plan sets for 10 queries derived from the TPC-H benchmark over generated data at scale factor 10. Then for each query, we compared the runtime of the plan chosen by the SQL Server query optimizer (SS) with the runtime of the plan that would be chosen by a modified version of this optimizer that was constrained to consider only the candidate plan sets enumerated by $\texttt{JPE}_\texttt{C}$ (SS + $\texttt{JPE}_\texttt{C}$). Each plan was run in SQL Server installed on an Intel Xeon 2.66 GHz machine with 8 GB RAM and two 700GB disks, and the results are shown in Table 1(a). The leftmost column shows the TPC-H based query and in parentheses, the number of tables joined by that query. The next two columns show the runtime of the plan chosen by SS for that query, and the amount of work ("Join Work Units"[1]) required by the optimizer to select that plan. The final two columns show the runtime chosen from the $\texttt{JPE}_\texttt{C}$-enumerated candidate set and the number of JWUs required to consider all plans in this set.

The results show that SS + $\texttt{JPE}_\texttt{C}$ chooses a better plan than the native SQL Server optimizer 30% of the time (and never chooses one that is worse), and does less work 70% of the time (and never does more work). The results are especially pronounced for the most complex 4 queries consisting of 6 or more tables where $\texttt{JPE}_\texttt{C}$ resulted in choosing a better plan 75% of the time (and on average, a factor of 2 faster) while doing less work 100% of the time (and on average, 35% of the work). These results are suggestive of the potential benefit from designing a targeted JPE.

3.2 Benchmarking and Profiling

The benchmarking described above required access to some of the internals of the SQL Server engine. For example, we relied on SQL Server's MEMO structure to determine the amount of work performed by SS and SS + $\texttt{JPE}_\texttt{C}$ to optimize a query. This makes the reported benchmark system-specific, and not a stand alone, independent benchmark as would be preferred.

To evaluate and profile JPEs developed, we will need to produce a benchmark that measures the quality of candidate plan sets that is independent of the cost model used to select final plans. Intuitively, this benchmark should assess the "goodness" of candidate sets by the degree to which they include good plans while excluding poor ones. Such a benchmark could have multiple metrics (e.g., size of candidate sets, best, average and worst runtimes of candidate plans, variation of candidate plan runtimes) but should balance these metrics in some meaningful way.

[1] A JWU characterizes the work that the SQL Server optimizer performs in processing one logical join of two tables. We use JWUs because SQL Server frequently prunes plans prior to consideration of complete plans and memorizes cost estimates. The count of JWUs shown is therefore the number of distinct pairs of tables whose logical joins are considered by the optimizer.

In general, the number of plans in the candidate set for a query is affected by the degree to which joins in that query are distinguished as belonging to different categories (and hence having different ranks). At one extreme, a JPE that identifies every join in the query as having unique rank will produce a candidate set consisting of a single plan. At the other extreme, a JPE that identifies every join as having the same rank will produce a candidate set consisting of all possible plans (excluding cartesian products). This observation leads to a natural way to "tune" a JPE specification: if a JPE tends to produce small candidate sets that miss good plans, it can be tuned by *merging* some categories so that they have equivalent rank, and if a JPE tends to produce large candidate sets that include bad plans, it can be tuned by *splitting* some categories so that more joins are differentiated.

While category merging and splitting are useful tuning strategies, the choices of *which* categories to merge or split is not always obvious. Thus, we plan for our development environment to include a tool that gathers statistics on *category prevalence*, whereby for some set of queries, it can be determined what percentage of joins contained in the queries belong to each category specified by a JPE. Such a tool could be used to find likely categories to merge or split depending on how much larger or smaller the desired candidate sets should be.

4. REMAINING COMPONENTS

In the previous section, we presented initial work on our development environment that supports the design of JPEs. Our development environment will similarly support the remaining components presented in Section 2, as well as the plan structures they process and the plan properties they exploit. In this section, we share some preliminary thoughts on this work in progress. Due to space constraints, we do not expand on the part of the development environment that is focused on the PPM which, because it simply substitutes physical plan operators for logical joins and table references, we believe it will have a fairly straightforward design.

4.1 Plans and Properties

Optimizer components rely on properties of plans in deciding what plans to construct or discard. For example, a pruner likely uses a *cost* property that estimates the execution cost of a plan as well as other "interesting" plan properties to decide which plans to keep and which to discard. In Section 3, we showed a JPE (JPE$_C$) that considered the *order* of data produced by a subplan in deciding what plans containing that subplan to consider.

Our proposed development environment supports the specification and generation of both the plan structures that are processed by optimizer components and the plan properties that are used by components in deciding how to process plans. By allowing developers to define their own plans, our development environment supports query languages that evolve over time and new plan operations as they are invented. By allowing them to define their own plan properties, new optimization strategies can be implemented that target novel architectures or applications.

Specification Language. Plan structures and properties are defined using an attribute grammar [4], which defines attributes for the *production rules* of a formal grammar, while attributes can also be synthesized through *attribute rules* and evaluated when an expression is processed by a compiler. In our grammar plan structures, operators and base relations are expressed as grammar symbols and their relationships and plan syntax as production rules. Properties are expressed as attributes and their evaluation models as attribute rules. The grammar offers a declarative language through which developers can specify: (i) statistical and architecture-specific properties (e.g., cardinality, distribution) (ii) property evaluation models and (iii) cost models. We demonstrate this with an example.

Let us assume a grammar that allows only for select and join operations implemented only by sequential scans and nested loop joins, respectively. Figure 4(a) shows a subset of the grammar's rules[2]. . Based on the production rules, a plan (`Plan`) can be either a logical (`LPlan`) or a physical (`LPlan`). `LPlan` can include only logical joins (`LJoin`), selections (`LSelect`) or base relations (`LTable`), while `PPlan` can include nested-loop joins (`PNLJoin`) or sequential scans (`PSeqScan`). A join takes as input two `Plan`s, a operator (`Op`) and the identifiers of its join attributes, while a select applies a operator (`Op`) on a plan. The attribute rules correspond to specific production rules and evaluate the cost and cardinality properties.

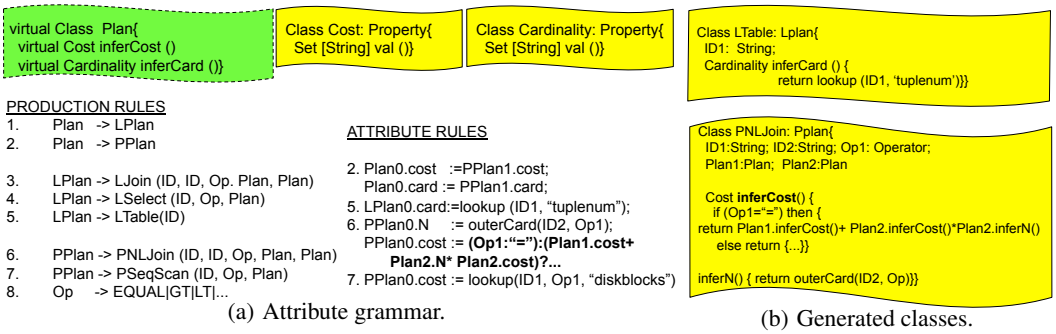
The cardinality of a base relation can be inferred by the external `lookup(ID, "tuplenum")` method that retrieves the set of tuples in the relation `ID` from the DBMS's metadata (Rule 5). The cost of sequential scans (Rule 7) equals to the disk blocks to be accessed in order to retrieve the required tuples. The cost of a nested-loop equi-join is defined as in System R, e.g., for an equi-join (`Op:"="`) it equals to the cost of the outer plan plus the cost of the inner plan multiplied the cardinality (N) of the outer relation tuples that satisfy the applicable predicates (`ID2, Op1`). Cost values are propagated from physical operators up to plans through simple copy-rules (Rule 2).

Class Generation. Given a grammar, a *code generator* produces a class library that implements the plan structure as well as the properties and their evaluation models. Figure 4(b) shows a subset of the classes generated for the example grammar and Figure 5(a) shows the class hierarchy. Each grammar symbol is defined by a separate class and the symbol on the right side of a production rule is a subclass of the symbol on the left side.

Plan structures and properties are expressed as *predefined, virtual* classes that are be further extended through a set of virtual methods. The generator creates a new set of *concrete* classes that inherit the functionality of their virtual super class and provide an implementation of the virtual methods. We illustrate these concepts through the examples in Figure 4(b) and 5(a).

For each attribute (e.g., cost) of the `Plan` symbol a concrete class is generated which inherits from the predefined virtual class `Property`. A `Plan` is implemented as a virtual class and it is replaced by the code generator to include an `inferPropertyX()` (e.g., `inferCost()`) virtual method for every attribute of the `Plan` symbol. Logical and physical plans are implemented as subclasses of the `Plan` class (and inherit its property inference methods), while logical and physical operators are concrete subclasses of the logical and physical plans, respectively. For example, `LTable` is a subclass of the `LSelect` and `NLJoin` is a subclass of `PPlan`. The code generator uses the Rule 5 and implements the `inferCard()` method in the class `LTable`, while based on Rule 7, the implementation of the `inferCost()` method for the nested-loop join is included in the class `PNLJoin`. The class hierarchy aligns with the hierarchical structure of a join plan and provides a bottom up evaluation of statistics. Hence, our grammar and its code generation tool offer a "natural" testbed that component developers can use in order to study how different property inference methods and cost models influence the effectiveness of the optimizer components.

[2]The language used here is inspired by the Ox attribute grammar parser [3].



(a) Attribute grammar. (b) Generated classes.

Figure 4: An example attribute grammar and its corresponding generated classes.

4.2 Pruner

A plan pruner accepts a set of physical plans produced by the PPM and returns the subset of them that satisfy specified properties (e.g., the "cheapest" plan according to a *cost* property or an "interesting" plan according to other properties such as *order*). Depending on when it is invoked by the Controller, it can return a single complete physical plan (e.g., the final join plan) or one or more partial physical plans to be further processed.

Specification and Generation. A natural approach to specifying a pruner is to identify a) a cost-based function that can be used to compare any two plans, and b) a set of properties that make a plan "interesting". Our intention is to leverage the plan properties specified within the attribute grammar specification of plans (Section 4.1) in both parts of the pruner specification.

An example is illustrated in Figure 5(b). The first class shown in this figure is the virtual class, `Pruner`, which declares two virtual methods (`Choose` and `Interesting`) and one concrete method `prune`. This class will be included in a class library extended by the Pruner generator tool. Any Pruner that is generated using this generator tool will be defined as a subclass of `Pruner` that includes concrete definitions of `Choose` and `Interesting` (e.g., class `CostBasedPruner` shown at the bottom of Figure 5(b).)

A Pruner specification needs to indicate how these virtual methods should be defined. The method, `Choose`, accepts two plans as input and should be defined to select which of these plans is 'preferred' according to the specified pruner. The method, `Interesting`, accepts a plan and a property and should be defined to return "`true`" if the plan is "interesting" with respect to that property. The concrete method `prune` in the class `Pruner` shows how methods `Choose` and `Interesting` are used for pruning query plans. This method accepts a set of plans as input (a set of plans produced by the PPM) and returns some subset of these plans as output. Thus, this method implements the pruning action performed by the pruner when invoked. As the method definition shows, the set of plans returned by this method consists of:

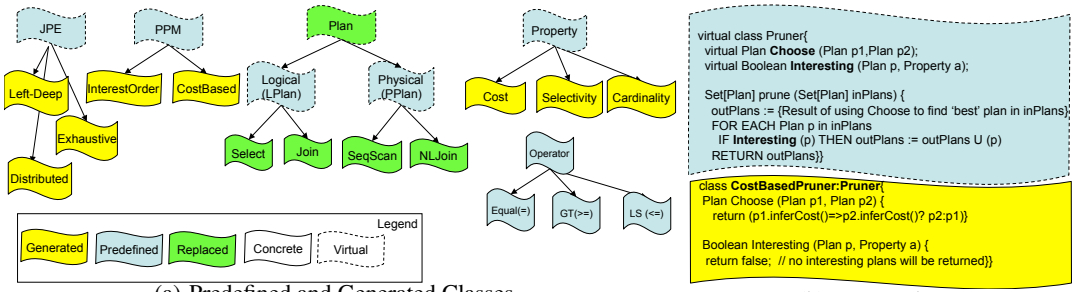
1. the "best" plan in the input set (i.e., the plan p returned by `Choose` when that method is invoked with p and any other plan in the input set as input), and
2. all additional plans from the input set for which the `Interesting` method returns "`true`".

Thus, when the `prune` method is invoked on a `CostBasedPruner` object with some set of plans, the result will be the single plan that is "best" according to method `Choose` (since this class defines no plan as 'interesting').

A pruner specification is likely to vary according to the underlying architecture of the system. For example, a data management system defined over a (centralized) client-server architecture might define `Choose` to use a cost model property that estimates the CPU and disk I/O costs of a plan, and define `Interesting` to return "`true`" for plans that have an inferred *order* attribute corresponding to the join attribute of a subsequent join (as this ordering allows the subsequent join to be performed as a merge join without having to sort the result produced by this plan). On the other hand, a data management system built over a distributed, shared nothing architecture might define `Choose` to use a cost model property that also estimates the network costs of a plan, and define `Interesting` to return "`true`" for plans that have an inferred *partition* attribute corresponding to the join attribute of a subsequent join, since this partitioning allows the subsequent join to be performed without having to repartition the result produced by this plan.

Benchmarking and Profiling. The benchmarking and profiling of a pruner should examine the effectiveness of its specified implementations of methods `Choose` and `Interesting`. The effectiveness of `Choose` depends on how accurate it is in selecting the better of any two plans. For any given set of n plans defined for the same query, this could be measured by comparing the $\frac{n\times(n-1)}{2}$ results of invoking `Choose` on distinct pairs of plans from this set, with comparisons of the execution times of the same pairs of plans. The effectivess of this method would then be measured by the percentage of times that `Choose` returns the plan with the better execution time given two distinct plans as input. Note that this metric is more forgiving than traditional cost model validation which determines the accuracy of predicted execution times of plans. A cost model needs only to predict the relative ranking of plans by their execution times and does not need to accurately estimate the times themselves. This allows for lightweight cost models, such as a model for a centralized DBMS that compares estimated I/O's in determining which of two plans is likely to have a lower execution time.

The effectiveness of a given pruner's implementation of the `Interesting` method could be ascertained by determining, for some set of queries, the number of plans that are chosen as "best" by the pruner (according to its `Choose` method) that would have been removed from consideration if not for some subplan being identified as interesting during pruning. Even a small percentage of "best" plans that were generated from interesting subplans might justify the use of the chosen property as interesting. A similar, but more elaborate metric could involve determining the percentage of times `Choose` (P, Q) returns P given some 'interesting' plan Q, but `Choose` (A_P, A_Q) returns A_Q such that A_Q is the plan that results from replacing the subplan P with Q in the plan A_P. The higher the value returned by this metric, the more effective is the interesting property in identifying a plans that is poor when considered in isolation but which is a subplan of some good plans.



(a) Predefined and Generated Classes. (b) Pruner class.

Figure 5: Example of a class hierarchy produced by the code generator and example of the pruner class.

4.3 Controller

A Controller is responsible for determining when during the optimization process, control is passed from the JPE to the PPM and Pruner. At one extreme, a *lazy* Controller could allow a JPE to enumerate all candidate join plans before passing them to the PPM (to generate equivalent physical plans) and the Pruner (to select a final plan). At the other extreme, an *eager* Controller could interrupt a JPE every time it orders a single join (i.e., converts an edge in the join graph) for any candidate join plan.

The choice in Controller design must balance the tradeoff between the resources required to optimize a query and the quality of chosen plans. Because a lazy controller constructs a set of *complete* physical plans before pruning any of them, the space requirements for optimization can be significant (as can optimizer time, though this is offset somewhat by memoization). On the other hand, an eager controller can aggressively prune *partial* physical plans before all of them have been enumerated, thereby using less space and time to optimize a query than a lazy controller, but often sacrificing the quality of the chosen plan given that some subplans may be unwisely pruned before they are evaluated within the context of a complete physical plan. (Note that the goal of retaining subplans with "interesting" properties is to prevent this from happening.)

Whether or not one can 'afford' a lazy controller will depend in large part on the size of candidate plan sets produced by a JPE. An optimizer with a lazy controller and an exhaustive enumerator will be heavily stressed when applied to complex queries. It is our belief that the smaller candidate plan sets enabled by our JPE design (Section 3) will make a lazy controller more feasible.

Specification and Generation. Any specification of a controller must indicate what are the *transfer points* during join plan enumeration where control should be transferred to the PPM and Pruner. Transfer points could be pre-determined (e.g., as with lazy or eager controllers) or could be defined by a predicate on the set of join graphs being processed by the JPE, such that a predicate would be evaluated after every edge conversion and control would be transferred to the PPM and Pruner when the predicate evaluated to "`true`". For example, a predicate might be specified to transfer control to the PPM and Pruner whenever the join graphs being processed exceeds some space threshold. This would prevent the optimizer from exceeding its memory allotment while allowing controllers to be as lazy as possible.

Benchmarking and Profiling. A controller benchmark should include metrics that describe how well the tradeoff between optimizer resource usage and quality of chosen plan has been balanced. In the ideal case, a controller design will be as eager as possible while still resulting in the optimizer selecting the same plan that would have been chosen with a lazy controller. Controllers could be evaluated with metrics that show, for some set of queries, the resource usage improvement (in terms of a percentage of space or time savings) and chosen plan dropoff (in terms of a percentage increase in plan execution time) versus a lazy controller that uses the same JPE, PPM and Pruner.

Profiling tools should assist in the design of an effective controller. For instance, a visualization tool might show the intermediate results produced by the JPE, PPM and Pruner every time control is transfered from one component to another. If the Pruner frequently fails to prune any of the physical plans produced by the PPM, this might suggest that the controller is too eager. On the other hand, if the Pruner eliminates all physical plans that share some subplan generated since the previous iteration, then the controller might be too lazy.

5. CONCLUSIONS

It is now commonly accepted that "one-size-does-not-fit-all" in data management systems, and it naturally follows that "one-design-does-not-fit-all" for the components of these systems. This means that it is likely that the designs of well-known data management system components (such as the query optimizer) will be reconsidered in light of the targeted applications and underlying architectures of these new systems. To support this effort, we have introduced a new *development environment* that we are designing that will support the rapid prototyping, evaluation and refinement of the key components of a query optimizer through the use of component generators, component-specific benchmarks and profiling tools. We have shown early results that include tools for designing and evaluating the join plan enumerator and we shared thoughts on how other optimizer components will be supported in the parts of the development environment whose design and implementation is work in progress.

6. REFERENCES

[1] D. J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008. PhD Thesis.

[2] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Exact Cardinality Query Optimization for Optimizer Testing. In *VLDB*, 2009.

[3] C. Fl, K. M. Bischoff, and K. M. Bischoff. Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C, 1993.

[4] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[5] S. Krishnamoorthy, A. K. Saple, and P. H. Achutharao. An integrated query optimization system for data grids. In *Compute '08*, 2008.

[6] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, 1990.

[7] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[8] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, 2007.

[9] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, 2009.