Grammar-like Functional Rules for Representing Query Optimization Alternatives

Guy M. Lohman
IBM Almaden Research Center
San Jose, CA 95120

Abstract

Extensible query optimization requires that the "repertoire" of alternative strategies for executing queries be represented as data, not embedded in the optimizer code. Recognizing that query optimizers are essentially expert systems, several researchers have suggested using strategy rules to transform query execution plans into alternative or better plans. Though extremely flexible, these systems can be very inefficient: at any step in the processing, many rules may be eligible for application and complicated conditions must be tested to determine that eligibility during unification. We present a constructive, "building blocks" approach to defining alternative plans, in which the rules defining alternatives are an extension of the productions of a grammar to resemble the definition of a function in mathematics. The extensions permit each token of the grammar to be parametrized and each of its alternative definitions to have a complex condition. The terminals of the grammar are base-level database operations on tables that are interpreted at run-time. The non-terminals are defined declaratively by production rules that combine those operations into meaningful plans for execution. Each production produces a set of alternative plans, each having a vector of properties, including the estimated cost of producing that plan. Productions can require certain properties of their inputs, such as tuple order and location, and we describe a "glue" mechanism for augmenting plans to achieve the required properties. We give detailed examples to illustrate the power and robustness of our rules and to contrast them with related ideas.

1. Introduction

Ever since the first query optimizers [WONG 76, SELI 79] were built for relational databases, revising the "repertoire" of ways to construct a procedural execution plan from a non-procedural query has required complicated and costly changes to the optimizer code itself. This has limited the repertoire of any one optimizer by discouraging or slowing experimentation with — and implementation of — all the new advances in relational technology, such as improved join methods [BABB 79, BRAT 84, DEWI 85], distributed query optimization [EPST 78, CHU 82, DANI 82, LOHM 85], semijoins [BERN 81], Bloomjoins [BABB 79, MACK 86], parallel joins on fragments [WONG 83], join indexes [HAER 78, VALD 87], dynamic creation of indexes [MACK 86], and many other variations of traditional processing strategies. The recent surge in interest in extensible database systems [STON 86, CARE 86, SCHW 86, BATO 86] has only exacerbated the burden on optimizers, adding the need to customize a database system for a particular class of applications, such as geographic [LOHM 83], CAD/CAM, or expert systems. Now optimizers must adapt to new access methods, storage managers, data types, user-defined functions, etc., all combined in novel ways. Clearly the traditional specification of all feasible strategies in the optimizer code cannot support such fluidity.

Perhaps the most challenging aspect of extensible query optimization is the representation of alternative execution strategies. Ideally, this representation should be readily understood and modified by the Database Customizer (DBC)[1]. Recognizing that query optimizers are expert systems, several authors have observed that rules show great promise for this purpose [ULLM 85, FREY 87, GRAE 87a]. Rules provide a high-level, *declarative* (i.e., non-procedural), and compact specification of legal alternatives, which may be input *as data* to the optimizer and traced to explain the origin of any execution plan. This makes it easy to modify the strategies without impacting the optimizer, and to encapsulate the strategies executable by a particular processor in a heterogeneous network. But how should rules represent alternative strategies? The EXODUS project [GRAE 87a, GRAE 87b] and Freytag [FREY 87] use rules to transform a given execution plan into other feasible plans. The NAIL! project [ULLM 85, MORR 86] employs "capture rules" to determine which of a set of available plans can be used to execute a query.

In this paper, we use rules to describe how to *construct* — rather than to *alter* or to *match* — plans. Our rules "compose" low-level database operations on tables (such as ACCESS, JOIN, and SORT) into higher-level operations that can be re-used in other definitions. These constructive, "building blocks" rules, which resemble the productions of a grammar, have two major advantages over plan transformation rules:

- **They are more readily understood**, because they enable the DBC to build increasingly complex plans from common building blocks, the details of which may be transparent to him; and
- **They can be processed more efficiently** during optimization, by simply finding the definition of any building block that is referenced, using a simple dictionary search, much as is done in macro expanders. By contrast, plan transformation rules usually must

1 We feel this term more accurately describes the role of adapting an implemented but extensible database system than does the term *Database Implementor (DBI)*, coined by Carey et al. [CARE 86].

examine a large set of rules and apply complicated conditions on each of a large set of plans generated thus far, in order to determine if that plan matches the pattern to which that rule applies. As new rules create new patterns, existing rules may have to add conditions that deal with those new patterns.

Our grammar-like approach is founded upon a few fundamental observations about query optimization:

- **All database operators consume and produce a common object — a table**, viewed as a stream of tuples that is generated by accessing a table [BATO 87a]. The output of one operation becomes the input of the next. Streams from individual tables are merged by joins, eventually into a single stream [FREY 87, GRAE 87a].

- **Optimizers construct legal sequences of such operators that are understood by an interpreter, the *query evaluator*.** In other words, the repertoire of legal plans is a *language* that might well be defined by a grammar.

- **Decisions made by the optimizer have an inherent sequence dependency** that limits the scope of subsequent decisions [BATO 87a, FREY 87]. For example, for a given plan, the order in which a given set of tables are joined must be determined before the access path for any of those tables is chosen, because the table order determines which predicates are eligible and hence might be applied by the access path of any table (commonly referred to as "pushing down the selection"). Thus, for any set of tables, the rules for ordering table accesses must precede those for choosing the access path of each table, and the former serve to limit significantly which of the latter rules are applicable.

- **Alternative plans may incorporate the same plan fragment,** whose alternatives need be evaluated only once. This further limits the rules generating alternatives to just the *new* portions of the plan.

- Unlike the simple pattern-matching of tokens to determine the applicability of productions in grammars, in query optimization **specifying the *conditions* under which a rule is applicable is usually harder than specifying the rule's *transformation*.** For example, a multi-column index can apply one or more predicates only if the columns referenced in the predicates form a prefix of the columns in the index. Assigning the predicates to be applied by the index is far easier to express than the condition that permits that assignment.

These observations prompted us to use "strategy" rules to construct legal nestings of database operators declaratively, much as the productions of a grammar construct legal sequences of tokens. However, our rules resemble more the definition of a function in mathematics or a rule in Prolog, in that the "tokens" of our grammar may be parametrized and their definition alternatives may have complex conditions. The reader is cautioned that the *application* — not the representation — is our claim to novelty. Logic programming uses rules to construct new relations from base relations [ULLM 85], whereas we are using rules to construct new operators from base operators that operate on tables.

Our approach is a general one, but we will present it in the context of its intended use: the Starburst prototype extensible database system, which is under development at the IBM Almaden Research Center [SCHW 86, LIND 87].

The paper is organized as follows. Section 2 first defines the end-product of optimization — plans. We describe what they're made of, what they look like, how our rules are used to construct all of them for a query. In Section 3, we associate properties with plans, and allow rules to impose requirements on the properties of their input plans. A set of possible rules for joins is given in Section 4 to illustrate the power of our rules to specify some of the most complicated strategies of existing systems, including several not addressed by other authors. Section 5 outlines how the DBC can make extensions to rules, properties, and database operators. Having thoroughly described our approach, we contrast it with related work in Section 6, and conclude in Section 7.

2. Plan Generation

In this section, we describe the form of our rules. We must first define what we want to produce with these rules, namely a query evaluation plan, and its constituents.

2.1. Plans

The basic object to be manipulated — and the class of "terminals" in our grammar — is a ***LOw-LEvel Plan OPerator (LOLEPOP)*** that will be interpreted by the query evaluator at run-time. LOLEPOPs are a variation of the relational algrebra (e.g., **JOIN, UNION**, etc.), supplemented with low-level operators such as **ACCESS, SORT, SHIP**, etc. [FREY 87]. Each LOLEPOP is viewed as a function that operates on 1 or 2 tables[2], which are parameters to that function, and produces a single table as output. A ***table*** can be either a table stored on disk or a "stream of tuples" in memory or a communication pipe. The **ACCESS** LOLEPOP converts a stored table to a stream of tuples, and the **STORE** LOLEPOP does the reverse. In addition to input tables, a LOLEPOP may have other parameters that control its operation. For example, one parameter of the **SORT** LOLEPOP is the set of columns on which to sort. Parameters may also specify a ***flavor*** of LOLEPOP. For example, different join methods having the same input parameter structure are represented by different flavors of the **JOIN** LOLEPOP; differences in input parameters would necessitate a distinct LOLEPOP. Parameters may be optional; for example, the **ACCESS** LOLEPOP may optionally apply a set of predicates.

A ***query evaluation plan*** (***QEP***, or ***plan***) is a directed graph of LOLEPOPs. An example plan is shown in Figure 1. Note that arrows point toward the source of the stream, not the direction in which tuples flow. This plan shows a sort-merge **JOIN** of DEPT as the outer table and EMP as the inner table. The DEPT stream is generated by an **ACCESS** to the stored table DEPT, then **SORT**ed into the order of column DNO for the merge-join. The EMP stream is generated by an **ACCESS** to the stored index on column EMP.DNO[3] that includes as one "column" the ***tuple identifier (TID)***. For each tuple in the stream, the **GET** LOLEPOP then uses the TID to get additional columns from its stored table: columns NAME and ADDRESS from EMP in this example.

Another way of representing this plan is as a nesting of functions [BATO 87a, FREY 87]:

JOIN (*sort-merge*, *DEPT.DNO=EMP.DNO*,
SORT(**ACCESS**(*DEPT*, {*DNO*, *MGR*}, {*MGR='Haas'*}), *DNO*),
GET(**ACCESS**(*Index on EMP.DNO*, {*TID*, *DNO*}, ϕ),
EMP, {*NAME*, *ADDRESS*}, ϕ))

This representation would be a lot more readable, and easier to construct, if we were to define intermediate functions D and E for the last two parameters to JOIN:

JOIN(*sort-merge*, D.*DNO*=E.*DNO*, D, E)

where

---

2 Nothing in the structure of our rules prevents LOLEPOPs from operating on any number of tables.

3 Actually, ACCESSes to base tables and to access methods such as this index use different flavors of ACCESS.

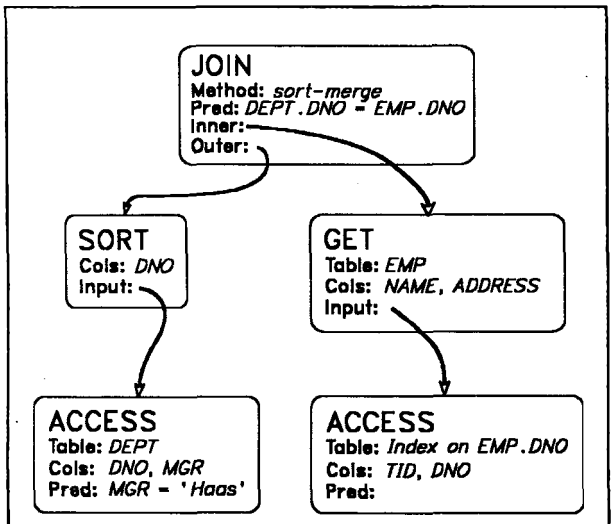

Figure 1: One potential query evaluation plan for the SQL query:

```
SELECT NAME, ADDRESS
FROM EMP E, DEPT D
WHERE E.DNO = D.DNO AND MGR='Haas'
```

$$D = \textbf{SORT}(\textbf{ACCESS}(DEPT, \{DNO, MGR\}, \{MGR='Haas'\}), DNO)$$

and

$$E = \textbf{GET}(\textbf{ACCESS}(Index\ on EMP.DNO, \{TID, DNO\}, \phi), EMP, \{NAME, ADDRESS\}, \phi)$$

If properly parametrized, these intermediate functions could be re-used for creating an ordered stream for *any* table, e.g.

$$\text{OrderedStream1}(T, C, P, order) = \textbf{SORT}(\textbf{ACCESS}(T, C, P), order)$$

and

$$\text{OrderedStream2}(T, C, P, order) = \textbf{GET}(\textbf{ACCESS}(a, \{TID\}, \phi), T, C, P) \quad \text{IF } order \sqsubseteq a$$

where T is the stored table (base table or base tables represented in a stored intermediate result) to be accessed, C is the set of columns to be accessed, P is the set of predicates to be applied, and "$order \sqsubseteq a$" means "the ordered list of columns of *order* are a prefix of those of access path a of T". Now it becomes apparent that OrderedStream1 and OrderedStream2 provide two alternative definitions for a single concept, an OrderedStream, in which the second definition depends upon the existence of a suitable access path:

$$\text{OrderedStream}(T, C, P, order) = \begin{cases} \textbf{SORT}(\textbf{ACCESS}(T, C, P), order) \\ \textbf{GET}(\textbf{ACCESS}(a, \{TID\}, \phi), T, C, P) \quad \text{IF } order \sqsubseteq a \end{cases}$$

This higher-level construct can now be nested within other functions needing an ordered stream, without having to worry about the details of how the ordered stream was created [BATO 87a]. It is precisely this train of reasoning that inspired the grammar-like design of our rules for constructing plans.

2.2. Rules

Executable plans are defined using a grammar-like set of parametrized production rules called ***STrategy Alternative Rules (STARs)*** that define higher-level constructs from lower-level constructs, in a way resembling common mathematical functions or a functional programming language [BACK 78]. A STAR defines a named, parametrized object (the "nonterminals" in our grammar) in terms of one or more ***alternative definitions***, each of which:

- may have a ***condition of applicability***, and
- defines a plan by referencing one or more LOLEPOPs or other STARs, specifying ***arguments*** for their parameters.

Arguments and conditions of applicability may reference constants, parameters of the STAR being defined, or other LOLEPOPs or STARs. For example, the intermediate functions OrderedStream1 and OrderedStream2, defined above, are examples of STARs with only one alternative definition, but OrderedStream has two alternative definitions. The first of these references the SORT LOLEPOP, whose first argument is a reference to the ACCESS LOLEPOP and whose second argument is the parameter *order*. The conditions of applicability for all the alternatives may either overlap or be exclusive. If they overlap, as they do for OrderedStream, then the STAR may return more than one plan.

In addition, we may wish to apply the function to every element of a set. For example, in OrderedStream2 above, any other index on EMP having DNO as its major column could achieve the desired order. So we need a STAR to generate an ACCESS plan *for each index i in that set I*:

$$\text{IndexAccess}(T) = \forall i \in I : \textbf{ACCESS}(i, \{TID\}, \phi)$$

Using rule IndexAccess in rule OrderedStream2 as the first argument should apply the GET LOLEPOP to each such plan, i.e., for each alternative plan returned by IndexAccess, the **GET** function will be referenced with that plan as its first argument. So **GET** (IndexAccess(*EMP*), *C*, *P*) will also return multiple plans. Therefore any STAR having overlapping conditions or referencing a multi-valued STAR will itself be multi-valued. It is easiest to treat all STARs as operations on the abstract data type ***Set of Alternative Plans for a stream (SAP)***, which consume one or two SAPs and are mapped (in the LISP sense [FREY 87]) onto each element of those SAPs to produce an output SAP. Set-valued parameters other than SAPs (such as the sets of columns *C* and predicates *P* above) are treated as a single parameter unless otherwise designated by the ∀ clause, as was done in the definition of IndexAccess.

2.3. Use and Implementation

As our functional notation suggests, the rule mechanism starts with the ***root STAR***, which is the "starting state" of our grammar. The root STAR has one or more alternative definitions, each of which may reference other STARs, which in turn may reference other STARs, and so on top down until a STAR is defined totally in terms of "terminals", i.e. LOLEPOPs operating on constants. Each reference of a STAR is evaluated by replacing the reference with its alternative definitions that satisfy the condition of applicability, and replacing the parameters of those definitions with the arguments of the reference. Unlike transformational rules, this substitution process is remarkably simple and fast, the fanout of any reference of a STAR is limited to just those STARs referenced in its definition, and alternative definitions may be evaluated in parallel. Therein lies the real advantage of STARs over transformational rules. The implementation of a prototype interpreter for STARs, including a very general mechanism for controlling the order in which STARs are evaluated, is described in [LEE 88].

Thus far in Starburst, we have sets of STARs for accessing individual tables and joins, but STARs may be defined for any new operation, e.g. outer join, and may reference any other STAR. The root STAR for joins is called JoinRoot, a possible definition of which appears in Section "4. Example: Join STARs", along with the STARs that it references. Simplified definitions of the single-table access STARs are given in [LEE 88]. For any given SQL query, we build plans bottom up, first referencing the AccessRoot STAR to build plans to access individual tables, and then repeatedly referencing the JoinRoot STAR to join plans that were generated earlier, until all tables have been joined. What constitutes a joinable pair of streams depends upon a compile-time parameter. The default is to give preference to those streams having an eligible join predicate linking them, as did System R and R*, but this can be overridden to also consider Cartesian products between two streams of small estimated cardinality. In addition, in Starburst we exploit all predicates that reference more than one table as join predicates. This generalization of System R's and R*'s "col1 = col2" join predicates, plus allowing plans to have composite inners (e.g., (A*B)*(C*D)) and Cartesian products (when the appropriate parameters are specified), significantly complicates the generation of legal join pairs and increases their number. However, a cheaper plan is more likely to be discovered among this expanded repertoire! We will address this aspect of query optimization in a forthcoming paper on join enumeration.

3. Properties of Plans

The concept of cost has been generalized to include all properties a plan might have. We next present how properties are defined and changed, and how they interact with STARs.

3.1. Description

Every table (either base table or result of a plan) has a set of ***properties*** that summarize the work done on the table thus far (as in [GRAE 87b], [BATO 87a],and [ROSE 87]) and hence are important to the cost model. These properties are of three types:

relational: the relational content of the plan, e.g. due to joins, projections, and selections

physical: the physical aspects of the tuples, which affect the cost but not the relational content, e.g. the order of the tuples

estimated: properties derived from the previous two as part of the cost model, e.g. estimated cardinality of the result and cost to produce it.

Examples of these properties are summarized in Figure 2. All properties are handled uniformly as elements of a ***property vector***, which can easily be extended to add more properties (see section 5).

Initially, the properties of stored objects such as tables and access methods are determined from the system catalogs. For example, for a table, the catalogs contain its constituent columns (COLS), the SITE at which it is stored [LOHM 85], and the access PATHS defined on it. No predicates (PREDS) have been applied yet, it is not a TEMPorary table, and no COST has been incurred in the query. The ORDER is "unknown" unless the table is known to store tuples in some order, in which case the order is defined by the ordered set of columns on which the tuples are ordered.

Each LOLEPOP changes selected properties, including adding cost, in a way determined by the arguments of its reference and the properties of any arguments that are plans. For example, **SORT** changes the ORDER of tuples to the order specified in a parameter. **SHIP** changes the SITE property to the specified site. Both LOLEPOPs add to the COST property of their input stream additional cost that depends upon the size of that stream, which is a function of its properties CARD and COLS. **ACCESS** changes a stored table to a memory-resident stream of tuples, but optionally can also subset columns (relational project) and apply predicates (relational select) that may be enumerated as arguments. The latter option will of course change the CARD property as well. These changes, including the appropriate cost and cardinality estimates, are defined in Starburst by a ***property function*** for each LOLEPOP. Each property function is passed the arguments of the LOLEPOP, including the property vector for arguments that are STARs or LOLEPOPs, and returns the revised property vector. Thus, once STARs are reduced to LOLEPOPs, the cost of any plan can be assessed by invoking the property function for successive LOLEPOPs. These cost functions are well established and validated [MACK 86], so will not be discussed further here.

- Relational (WHAT)

TABLES	Set of tables accessed
COLS	Set of columns accessed
PREDS	Set of predicates applied

- Physical (HOW)

ORDER	Ordering of tuples (an ordered list of columns)
SITE	Site to which tuples delivered
TEMP	"True" if materialized in a temporary table
PATHS	Set of available access paths on (set of) tables, each element an ordered list of columns

- Estimated (HOW MUCH)

CARD	Estimated number of tuples resulting
COST	Estimated cost (total resources, a linear combination of I/O, CPU, and communications costs [LOHM 85])

Figure 2: Example properties of a plan.

3.2. Required vs. Available Properties

A reference of a STAR or LOLEPOP, especially for certain join methods, may require certain properties for its arguments. For example, the merge-join requires its input table streams to be ordered by the join columns, and the nested-loop join requires the inner table's access method to apply the join predicate as though it were a single-table predicate ("pushes the selection down"). Dyadic LOLEPOPs such as **GET**, **JOIN**, and **UNION** require that the SITE of both input streams be the same.

In the previous section, we constructed a STAR for an OrderedStream, where the desired order was a parameter of that STAR. Clearly we could require a particular order by referencing OrderedStream with the required order as the corresponding argument. The problem is that we may simultaneously require values for any of the 2^n combinations of n properties, and hence would have to have a differently-named STAR for each combination. For example, if the sort-merge **JOIN** in the example is to take place

at SITE=x, then we need to define a SitedOrderedStream that has parameters for SITE and ORDER and references in its definition SHIP LOLEPOPs to send any stream to SITE x, as well as a SitedStream, an OrderedStream, and a STREAM. Actually, SitedOrderedStream subsumes the others, since we can pass nulls for the properties not required. But in general, every STAR will need this same capability to specify some or all of the properties that might be required by referencing STARs as parameters. Much of the definition of each of these STARs would be redundant, because these properties really are orthogonal to what the stream produces. In addition, we often want to find the *cheapest* plan that satisfies the required properties, even if there is a plan that naturally produces the required properties. For example, even though there is an index EMP.DNO by which we can access EMP in the required DNO order, it might be cheaper, if EMP were not ordered by DNO, to access EMP sequentially and sort it into DNO order.

We therefore factor out a separate mechanism called ***Glue***, which can be referenced by any STAR and which:

1. checks if any plans exist for the required relational properties (TABLES, COLS, and PREDS), referencing the top-most STAR with those parameters if not;
2. adds to any existing plan "Glue" operators as a "veneer" to achieve the required properties (for example, a **SORT** LOLEPOP can be added to change the tuple ORDER, or a **SHIP** LOLEPOP to change the SITE); and
3. either returns the cheapest plan satisfying the requirments or (optionally) all plans satisfying the requirements.

In fact, Glue can be specified using STARs, and Glue operators can be STARs as well as LOLEPOPs, as described in [LEE 88].

Required properties in the STAR reference are enclosed in square brackets next to the affected SAP argument, to associate the required properties with the stream on which they are imposing requirements. Different properties may be required by references in different STARs; the requirements are accumulated until Glue is referenced. This will be illustrated in the next section.

An example of this Glue mechanism is shown in Figure 3. In this example, we assume that table DEPT is stored at SITE=N.Y., but the STAR requires DEPT to be delivered to SITE=L.A. in DNO order. None of the available plans meets those requirements. The first available plan must be augmented with a **SHIP** LOLEPOP to change the SITE property from N.Y. to L.A. The second plan, a simple **ACCESS** of DEPT, must be both **SORT**ed and **SHIP**ped The third plan, perhaps created by an earlier reference of Glue that didn't have the ORDER requirement, has already added a **SHIP** to plan 2 to get it to L.A., but still needs a **SORT** to achieve the ORDER requirement.

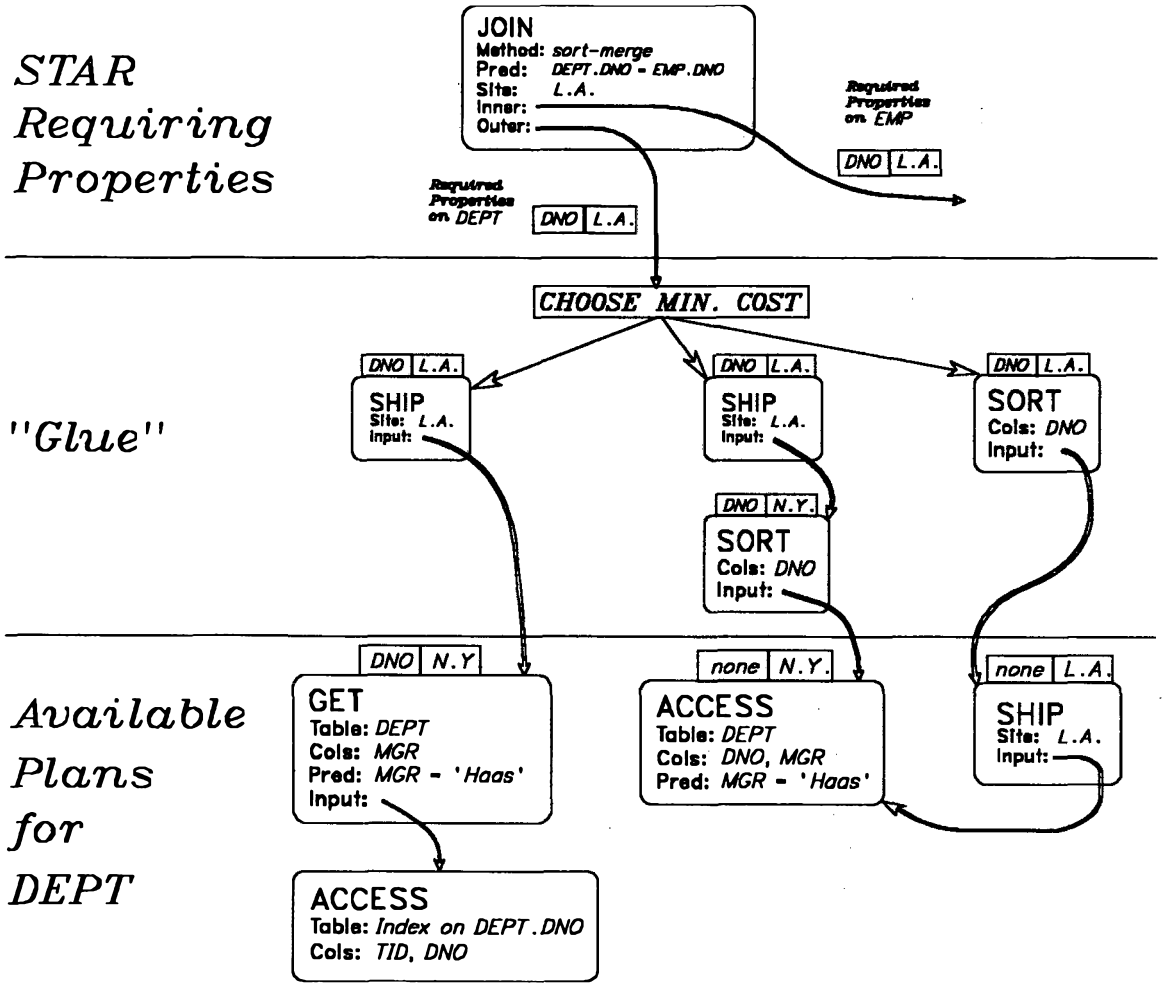

Figure 3: Example of "Glue" mechanism injecting "Glue" operators to match plans to required properties, and choosing the cheapest. Only two properties, order and site, are shown here, as "ears" on top of the top-most LOLEPOP for each plan.

4. Example: Join STARs

To illustrate the power of STARs, in this section we discuss one possible set of STARs for generating the join strategies of the R* optimizer (in Sections 4.1 - 4.4), plus several additional strategies such as

- composite inners (Sections 4.1 and 4.3),
- new access methods (Section 4.5.2),
- new join methods (Section 4.4),
- dynamic creation of indexes on intermediate results (Section 4.5.3),
- materialization of inner streams of nested-loop joins to force projection (Section 4.5.2).

Although there may be better ways within our STAR structure to express the same set of strategies, the purpose of this section is to illustrate the full power of STARs. Some of the strategies (e.g., hash joins) have not yet been implemented in Starburst; they are included merely for illustrating what is involved in adding these strategies to the optimizer.

These STARs are by no means complete: we have intentionally simplified them by removing parameters and STARs that deal with subqueries treated as joins, for example. The reader is cautioned against construing this omission as an inability to handle other cases; on the contrary, it illustrates the flexibility of STARs! We can construct, but have omitted for brevity, additional STARs for

- sorting TIDs taken from an unordered index in order to order I/O accesses to data pages,
- ANDing and ORing of multiple indexes for a single table,
- treating subqueries as joins having different quantifier types (i.e., generalizing the predicate calculus quantifiers of ALL and EXISTS to include the FOR EACH quantifier for joins and the UNIQUE quantifier for scalar ("=") subqueries),
- filtration methods such as semi-joins and Bloom-joins.

We believe that any desired strategy for non-recursive queries will be expressible using STARs, and are currently investigating what difficulties, if any, arise with recursive queries and multiple execution streams resulting from table partitioning [BATO 87a].

In these definitions, for readability we denote ***exclusive alternative definitions*** by a left *curly brace* and ***inclusive alternative definitions*** by a left *square bracket*. In practice, no distinction is necessary. In all examples, we will write non-terminals (STAR names) in RegularMixedCase, parameters in *italics* (those which may be sets are denoted by capital letters), and terminals in bold, with LOLEPOPs distinguished by **BOLD CAPITAL LETTERS**. Required properties are written in **small bold letters** and surrounded by a pair of [square brackets]. For brevity, we have had to shorten names, e.g., "JMeth" should read "JoinMethod". The function "$\chi(\bullet)$" denotes "columns of $(\bullet)$", where $\bullet$ can be a set of tables, an index, etc. We assume the existence of the basic set functions of $\in, \cap, \subseteq, -$ (set difference), etc.

STARs are defined here top down (i.e., a STAR referenced by any STAR is defined after its reference), which is also the order in which they will be referenced. We start with the root STAR, JoinRoot, which is referenced for a given set of parameters:

- table (quantifier) sets *T1* and *T2* (with no order implied)
- the set of (newly) eligible predicates, *P*

Suppose, for example, that plans for joining tables X and Y and for accessing table Z had already been generated, so we were ready to construct plans for joining X*Y with Z. Then JoinRoot would be referenced with $T1 = \{X,Y\}$, $T2 = \{Z\}$, and $P = \{X.g = Z.m, Y.h = Z.n\}$.

4.1. Join Permutation Alternatives

$$\text{JoinRoot}(T1, T2, P) = \left[\begin{array}{l}\text{PermutedJoin}(T1, T2, P)\\ \text{PermutedJoin}(T2, T1, P)\end{array}\right.$$

The meaning of this STAR should be obvious: either table-set *T1* or table-set *T2* can be the outer stream, with the other table-set as the inner stream. Both are possible alternatives, denoted by an inclusive (square) bracket. Note that we have no conditions on either alternative; to exclude a *composite inner* (i.e., an inner that is itself the result of a join), we could add a condition restricting the inner table-set to be one table.

This simple STAR fails to adequately tax the power of STARs, and thus resembles the comparable rule of transformational approaches. However, note that since none of the STARs referenced by JoinRoot or any of its descendants will reference JoinRoot, there is no danger of this STAR being invoked again and "undoing" its effect, as there is in transformational rules [GRAE 87a].

4.2. Join-Site Alternatives

$$\text{PermutedJoin}(T1, T2, P) = \begin{cases}\text{SitedJoin}(T1, T2, P) & \text{IF local query}\\ \forall s \in \sigma: \text{RemoteJoin}(T1, T2, P, s) & \text{OTHERWISE}\end{cases}$$

$$\text{RemoteJoin}(T1, T2, P, s) = \text{SitedJoin}(T1[\textbf{\textit{site}} = s], T2[\textbf{\textit{site}} = s], P)$$

where

$\sigma \equiv$ set of sites at which tables of the query are stored, plus the query site

This STAR generates the same join-site alternatives as R* [LOHM 84], and illustrates the specification of a required property. Note that Glue is not referenced yet, so the required site property accumulates on each alternative until it is. The interpretation is:

1. If all tables (of the query) are located at the query site, go on to SitedJoin, i.e., bypass the RemoteJoin STAR which dictates the join site.
2. Otherwise, require that the join take place at one of the sites at which tables are stored or the query originated.

If a site with a particularly efficient join engine were available, then that site could easily be added to the definition of σ.

4.3. Store Inner Stream?

$$\text{SitedJoin}(T1, T2, P) = \begin{cases}\text{JMeth}(T1, T2[\textbf{\textit{temp}}], P) & \text{IF C1}\\ \text{JMeth}(T1, T2, P) & \text{OTHERWISE}\end{cases}$$

where

$C1 \equiv$ IF $|T2| > 1$ OR $T2[\textbf{\textit{site}}] \neq T2![\textbf{\textit{site}}]$

Again, this simple STAR has an obvious interpretation, although the condition C1 is a bit complicated:

1. IF the inner stream (*T2*) is a composite, or its site is not the same as its required site ($![\textbf{\textit{site}}]$), then dictate that it be stored as a temp and call JMeth.
2. OTHERWISE, reference JMeth with no additional requirements.

Note that if the second disjunct of condition C1 were absent, there would be no reason that this STAR couldn't be the parent

(referencer) of the previous STAR, instead of vice versa. As written, SitedJoin exploits decisions made in its parent STAR, PermutedJoin. A transformational rule would either have to test if the site decision were made yet, or else inject the temp requirement redundantly in every transformation that dictated a site.

4.4. Alternative Join Methods

```
JMeth(T1, T2, P) =

  JOIN(NL, Glue(T1, φ), Glue(T2, JP∪IP), JP, P−(JP∪IP))
  JOIN( MG, Glue(T1[order = χ(SP)∩χ(T1)], φ),
            Glue(T2[order = χ(SP)∩χ(T2)], IP),
            SP, P−(IP∪SP)                    ) IF SP≠φ
where

P  ≡  all eligible predicates
JP ≡  join predicates (multi-table, no ORs or
      subqueries, etc., but expressions OK)
SP ≡  sortable predicates
   =  { p∈JP of form 'col1 op col2', where
      col1∈χ(T1) & col2∈χ(T2) or vice versa }
IP ≡  predicates eligible on the inner only,
      i.e., predicates p such that χ(p) ⊆ χ(T2)
```

This STAR references two alternative join methods, both represented as references of the JOIN LOLEPOP with different parameters:

1. the join method (flavor of **JOIN**),
2. the outer stream and any required properties on that stream,
3. the inner stream and any required properties on that stream,
4. the join predicate(s) applicable by that join method (needed for the cost equations),
5. any residual predicates to apply *after* the join.

The two join methods here are:

1. **Nested-Loop (NL) Join,** which can always be done. For each outer tuple instance, columns of the join predicates (JP) in the outer are instantiated to convert each JP to a single-table predicate on the inner stream[4]. These and any predicates on just the inner (IP) are "pushed down" to be applied by the inner stream, if possible. Any multi-table predicates that don't qualify as join predicates must be applied as residual predicates. Note that the predicates to be applied by the inner stream are *parameters, not required attributes.* This forces Glue to re-reference the single-table STARs to generate plans that *exploit* the converted JP predicates rather than *retrofitting* a **FILTER** LOLEPOP to existing plans that applied only the IP predicates.

2. **Merge (MG) Join:** If there are sortable predicates (SP), dictate that both inner and outer be sorted on their columns of SP. Note that the merge join, unlike the nested-loop join, applies the sortable predicates as part of the join itself, pushing down to the inner stream only the single-table predicates on the inner (IP). The JOIN LOLEPOP in Figure 1, for example, would be generated by this alternative. As before, remaining multi-table predicates must be applied by JOIN as residuals after the join.

Glue will first reference the STARs for accessing the given table(s), applying the given predicate(s), if no plans exist for those parameters. In Starburst, a data structure hashed on the tables and predicates facilitates finding all such plans, if they exist. Glue then adds the necessary operators to each of these plans, as described in the previous section. Simplified STARs for Glue, which this STAR references, and for accessing stored tables, which Glue references, are given in [LEE 88].

4.5. Additional Join Methods

Suppose now we wanted to augment the above alternatives with additional join methods. All of the following alternative definitions would be added to the right-hand side of the above STAR (JMeth).

4.5.1. Hash Join Alternative

The hash join has shown promising performance [BABB 79, BRAT 84, DEWI 85]. We assume here a hash-join flavor (**HA**) that atomically bucketizes both input streams and does the join on the buckets.

```
JOIN(HA, Glue(T1, φ), Glue(T2, IP), HP, P−IP)   IF HP≠φ
where
   HP ≡ hashable predicates
      ≡ { p∈JP of form 'expr(χ(T1)) = expr(χ(T2))'}
```

As in the merge join, only single-table predicates can be pushed down to the inner. Note that all multi-table predicates (P-IP) — even the hashable predicates (HP) — remain as residual predicates, since there may be hash collisions. Also note that the set of hashable predicates HP contains some predicates not in the set of sortable predicates SP (expressions on any number of columns in the same table), and vice versa (inequalities).

An alternate (and probably preferable) approach would be to add a **bucketized** property to the property vector and a LOLEPOP to achieve that property, so that any join method in the JMeth STAR could perform the join in parallel on each of the bucketized streams, with appropriate adjustments to its cost.

4.5.2. Forcing Projection Alternative

To avoid expensive in-memory copying, tuples are normally retained as pages in the buffer just as they were **ACCESS**ed, until they are materialized as a temp or **SHIP**ped to another site. Therefore, in nested-loop joins it may be advantageous to materialize (**STORE**) the selected and projected inner and re-**ACCESS** it before joining, whenever a very small percentage of the inner table results (i.e., when the predicates on the inner table are quite selective and/or only a few columns are referenced). Batory suggests the same strategy whenever the inner "is generated by a complex expression" [BATO 87a]. The following forces that alternative:

```
JOIN( NL, Glue(T1, φ),
          TableAccess(Glue(T2[temp], IP), *, JP),
          JP, P−(IP∪JP)                          )
```

This JMeth alternative accesses the inner stream (*T2*), applying only the single-table predicates (IP), and forcing Glue to **STORE** the result in a temp (permanently stored tables are not considered temps initially). All columns (*) of the temp are then re-accessed, re-using the STAR for accessing any stored table, TableAccess. Note that the STAR structure allows us to specify that the join predicates (JP) can be pushed down only to this access, to prevent the temp from being re-materialized *for each outer tuple!*

---

4 Ullman has coined the term "sideways information passing" [ULLM 85] for this conversion of join predicates to single-table predicates by instantiating one side of the predicate, which was done in System R [SELI 79].

```
TableAccess(T, C, P) =

  { ACCESS(Heap, T, C, P)      IF StMgr(T) = 'heap'
  { ACCESS(BTree, T, C, P)     IF StMgr(T) = 'B-tree'
```

A TableAccess can be one (and only one) of the following flavors of ACCESS, depending upon the type of storage manager (StMgr) used, as described in [LIND 87]:

1. A physically-sequential ACCESS of the pages of table *T*, if the storage manager type of *T* is 'heap', or
2. A B-Tree type ACCESS of table *T*, if the storage manager type of *T* is 'B-tree',

retrieving columns *C* and applying predicates *P*. By now it should be apparent how easily alternatives for additional storage manager types could be added to this STAR alone, and affect all STARs that reference TableAccess.

4.5.3. Dynamic Indexes Alternative

The nested-loop join works best when an index on the inner table can be used to limit the search of the inner to only those tuples satisfying the join and/or single-table predicates on the inner. Such an index may not have been created by the user, or the inner may be an intermediate result, in which case no auxiliary access paths such as an index are normally created. However, we can force Glue to create the index as another alternative. Although this sounds more expensive than sorting for a merge join, it saves sorting the outer for a merge join, and will pay for itself when the join predicate is selective [MACK 86].

```
JOIN ( NL, Glue(T1, φ),
          Glue(T2[paths ⊇ IX], XP∪IP), XP − IP, P − (XP ∪ IP) )
where
   XP ≡ indexable multi-table predicates
      = { p ∈ JP of form 'expr(χ(T1)) op T2.col' }
   IX ≡ columns of indexable predicates
      = (χ(IP) ∪ χ(XP)) ∩ χ(T2), '=' predicates first
```

This alternative forces Glue to make sure that the access paths property of the inner contains an index on the columns that have either single-table (IP) or indexable (XP) predicates, ordered so that those involved in equality predicates are applied first. If this index needs to be created, the STARs implementing Glue will add **[*order*]** and **[*temp*]** requirements to ensure the creation of a compact index on a stored table. As in the nested-loop alternative, the indexable multi-table predicates "pushed down" to the inner are effectively converted to single-table predicates that change for each outer tuple.

5. Extensibility — What's Really Involved

Here we discuss briefly the steps required to change various aspects of the optimizer strategies, in order to demonstrate the extensibility and modularity of our STAR mechanism.

Easiest to change are the STARs themselves, when an existing set of LOLEPOPs suffices. If the STARs are treated as input data to a rule interpreter, then new STARs can be added to that file without impacting the Starburst system code at all [LEE 88]. If STARs are compiled to generate an optimizer (as in [GRAE 87a, GRAE 87b]), then updates of the STARs would be followed by a re-generation of the optimizer. In either case, any STAR having a condition not yet defined would require defining a C function for that condition, compiling that function, and relinking that part of the optimizer to Starburst. Note that we assume that the DBC specifies the STARs correctly, i.e. without infinite cycles or meaningless sequences of LOLEPOPs. An open issue is how to verify that any given set of STARs is correct.

Less frequently, we may wish to add a new LOLEPOP, e.g. **OUTERJOIN**. This necessitates defining and compiling two C functions: a run-time execution routine that will be invoked by the query evaluator, and a property function for the optimizer to specify the changes to plan properties (including cost) made by that LOLEPOP. In addition, STARs must be added and/or modified, as described above, to reference the LOLEPOP under the appropriate circumstances.

Probably the least likely and most serious alterations occur when a property is added (or changed in any way) in the property vector. Since the default action of any LOLEPOP on any property is to leave the input property unchanged, only those property functions that reference the new property would have to be updated, recompiled, and relinked to Starburst. By representing the property vector as a self-defining record having a variable number of fields, each of which is a property, we can insulate unaffected property functions from any changes to the structure of the property vector. STARs would be affected only if the new property were required or produced by that STAR.

6. Related Work

Some aspects of our STARs resemble features of earlier work, but there are some important differences. As we mentioned earlier, our STARs are inspired by functional programming concepts [BACK 78]. A major difference is that our "functions" (STARs) can be multi-valued, i.e. a *set* of alternative objects (plans). The other major inspiration, a production of a grammar, does not permit a condition upon alternative expansions of a non-terminal: it either matches or it doesn't (and the alternatives must be exclusive). Hoping to use a standard compiler generator to compile our STARs, we investigated the use of partially context-sensitive W-grammars [CLEA 77] for enforcing the "context" of required properties, but were discouraged by the same combinatorial explosion of productions described above when many properties are possible. Koster [KOST 71] has solved this using a technique similar to ours, in which a predicate called an "affix" (comparable to our condition of applicability) may be associated with each alternative definition. He has shown affix grammars to be Turing complete. In addition, grammars are typically used in a parser to find *just one* expansion to terminals, whereas our goal is to construct *all* such expansions. Although a grammar can be used to construct all legal sequences, this set may be infinite [ULLM 85].

The transformational approach of the EXODUS optimizer [GRAE 87a, GRAE 87b] uses C functions for the IF conditions and expresses the alternatives in rules, as do we, but then compiles those rules and conditions using an "optimizer generator" into executable code. Given one initial plan, this code generates all legal variations of that plan using two kinds of rules: transformation rules to define alternative transformations of a plan, and implementation rules to define alternative methods for implementing an operator (e.g., nested-loop and sort-merge algorithms for implementing the JOIN operator). Our approach does not require an initial plan, and has only one type of rule, which permits us to express interactions between transformations and methods. Our property functions are indistinguishable from Graefe's property and

cost functions, although we have identified more properties than any other author to date. Graefe does not deal with the need of some rules (e.g. merge join) to require certain properties, as discussed in Section 3.2 and illustrated in Sections 4.2 - 4.4, 4.5.2, and 4.5.3. Although Graefe re-uses common subplans in alternative plans, transformational rules may subsequently generate alternatives and pick a new optimal plan for the subplan, forcing re-estimation of the cost of every plan that has already incorporated that subplan. Our building blocks approach avoids this problem by generating all plans for the subplan before incorporating that subplan in other plans, although Glue may generate some new plans having different properties and/or parameters. And while the structure of our STARs does not preclude compilation by an optimizer generator, it also permits interpreting the STARs by a simple yet efficient interpreter during optimization, as was done in our prototype. Interpretation saves re-compiling the optimizer component every time a strategy is added or changed, and also allows greater control of the order of evaluation. For example, depending upon the value of a STAR's parameter, we may never have to construct entire subtrees within the decision tree, but a compiled optimizer must contain a completely general decision tree for all queries.

Freytag [FREY 87] proposes a more LISP-like set of transformational rules that starts from a non-procedural set of parameters from the query, as do we, and transforms them into all alternative plans. He points to the EXODUS optimizer generator as a possible implementation, but does not address several key implementation issues such as his ellipsis ("...") operator, which denotes any number of expressions, e.g.:

$$((\text{JOIN } T_1(...T_2...)) \rightarrow (\text{JOIN } T_1(... \ ...)\, T_2))$$

And the ORDER and SITE properties (only) are expressed as functions, which presumably would have to be re-derived each time they were referenced in the conditions. Freytag does not exploit the structure of query optimization to limit what rules are applicable at any time and to prevent re-application of the same rules to common subplans shared by two alternative plans, although he suggests the need to do so.

Rosenthal and Helman [ROSE 87] suggest specifications for "well-formed" plans, so that transformational rules can be verified as valid if they transform well-formed plans to well-formed plans. Like Graefe, they associate properties with plans, viewed as predicates that are true about the plan. Alternative plans producing the same intermediate result with the same properties converge on "data nodes", on which "transformations that insert unary operators...are more naturally applied". An operator is then well-formed if any input plan satisfying the required input properties produces an output plan that satisfies the output properties. The paper emphasizes representations for verifiability and search issues, rather than detailing mechanisms (1) to construct well-formed transformations, (2) to match input data nodes to output data nodes (corresponding to our Glue), and (3) to recalculate the cost of all plans that share (through a common data node) a common subplan that is altered by a transformation.

Probably the closest work to ours is Batory's "synthetic" architecture for the entire GENESIS extensible database system (not just the query optimizer [BATO 87b]), in which "atoms" of "primitive algorithms" are composed by functions into "molecules", in layers that successively add implementation details [BATO 87a]. Developed concurrently and independently, Batory's functional notation closely resembles STARs, but is presented and implemented as rewrite (transformational) rules that are used to construct and compile the complete set of alternatives *a priori* for a given optimizer, after first selecting from a catalog of available algorithms those desired to implement operators for each layer. At the highest layer, for example, the DBC chooses from many optimization algorithms (e.g., depth-first vs. breadth-first), while the choices at the lowest layers correspond to our flavors of LOLEPOPs or Graefe's methods. The functions that compose these operations do not explicitly permit conditions on the alternative definitions, as do we; Batory considers them unnecessary when rules are constructed properly, but alludes to them in comments next to some alternatives and in a footnote. Inclusive alternatives automatically become arguments of a CHOOSE_CHEAPEST function during the composition process. The rewrite rules include rules to match properties (which he calls characteristics) even if they are unneeded, e.g. a SORT may be applied to a stream that is already ordered appropriately by an index, as well as rules to simplify the resulting compositions and eliminate any such unnecessary operations. By treating the stored vs. in-memory distinction as a property of streams, and by having a general-purpose Glue mechanism, we manage to factor out most of these redundancies in our STARs. Although clearly relevant to query optimization, Batory's larger goal was to incorporate an encyclopedic array of known query processing algorithms within his framework, including operators for splitting, processing in parallel, and assembling horizontal partitions of tables.

7. Conclusions

We have presented a grammar for specifying the set of legal strategies that can be executed by the query evaluator. The grammar composes low-level database operators (LOLEPOPs) into higher-level constructs using rules (STARs) that resemble the definition of functions: they may have alternative definitions that have IF conditions, and these alternative definitions may, in turn, reference other functions that have already been defined. The functions are parametrized objects that produce one or more alternative plans. Each plan has a vector of properties, including the cost to produce that plan, which may be altered only by LOLEPOPs. When an alternative definition requires certain properties of an input, "Glue" can be referenced to do "impedance matching" between the plans created thus far and the required properties by injecting a veneer of Glue operators.

We have shown the power of STARs by specifying some of the strategies considered by the R* system and several additional ones, and believe that any desired extension can be represented using STARs. We find our constructive, "building-blocks" grammar to be a more natural paradigm for specifying the "language" of legal sequences of database operators than plan transformational rules, because they allow the DBC to build higher levels of abstraction from lower-level constructs, without having to be aware of how those lower-level constructs are defined. And unlike plan transformational rules, which consider all rules applicable at every iteration and which must do complicated unification to determine applicability, referencing a STAR triggers in an obvious way only those STARs referenced in its definition, just like a macro expander. This limited fanout of STARs should make it possible to achieve our goal of expressing alternative optimizer strategies as data and still use these rules to generate and evaluate the cost of a large number of plans within a reasonable amount of time.

8. Acknowledgements

We wish to acknowledge the contributions to this work by several colleagues, especially the Starburst project team. We particularly benefitted from lengthy discussions with — and suggestions by — Johann Christoph Freytag (now at the European Community Research Center in Munich), Laura Haas, and Kiyoshi Ono (visiting from the IBM Tokyo Research Laboratory). Laura Haas, Bruce Lindsay, Tim Malkemus (IBM Entry Systems Division in Austin, TX), John McPherson, Kiyoshi Ono, Hamid Pirahesh, Irv Traiger, and Paul Wilms constructively critiqued an earlier draft of this paper, improving its readability significantly. We also thank the referees for their helpful suggestions.

Bibliography

[BABB 79] E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Systems* **4,1** (1979) pp. 1-29.

[BATO 86] D.S. Batory et al., GENESIS: An Extensible Database Management System, *Tech. Report TR-86-07* (Dept. of Comp. Sci., Univ. of Texas at. To appear in IEEE Trans. on Software Engineering.

[BATO 87a] D.S. Batory, A Molecular Database Systems Technology, *Tech. Report TR-87-23* (Dept. of Comp. Sci., Univ. of Texas at.

[BATO 87b] D. Batory, Extensible Cost Models and Query Optimization in GENESIS, *IEEE Database Engineering* **10,4** (Nov. 1987).

[BACK 78] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Comm. ACM* **21,8** (Aug. 1978).

[BERN 81] P. Bernstein and D.-H. Chiu, Using Semi-Joins to Solve Relational Queries, *Journal ACM* **28,1** (Jan. 1981) pp. 25-40.

[BRAT 84] K. Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Procs. of the Tenth International Conf. on Very Large Data Bases (Singapore),* **Morgan Kaufmann Publishers** (Los Altos, CA, 1984) pp. 323-333.

[CARE 86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, and M. Muralikrishna, The Architecture of the EXODUS Extensible DBMS: a Preliminary Report, *Procs. of the International Workshop on Object-Oriented Database Systems* (Asilomar, CA, Sept. 1986).

[CHU 82] W.W. Chu and P. Hurley, Optimal Query Processing for Distributed Database Systems, *IEEE Trans. on Computers* **C-31,9** (Sept. 1982) pp. 835-850.

[CLEA 77] J.C. Cleaveland and R.C. Uzgalis, *Grammars for Programming Languages,* **Elsevier North-Holland** (New York, 1977).

[DANI 82] D. Daniels, P.G. Selinger, L.M. Haas, B.G. Lindsay, C. Mohan, A. Walker, and P. Wilms, An Introduction to Distributed Query Compilation in R*, *Procs. Second International Conf. on Distributed Databases* (Berlin, September 1982). Also available as IBM Research Report RJ3497, San Jose, CA, June 1982.

[DEWI 85] D.J. DeWitt and R. Gerber, Multiprocessor Hash-Based Join Algorithms, *Procs. of the Eleventh International Conf. on Very Large Data Bases (Stockholm, Sweden),* **Morgan Kaufmann Publishers** (Los Altos, CA, September 1985) pp. 151-164.

[EPST 78] R. Epstein, M. Stonebraker, and E. Wong, Distributed Query Processing in a Relational Data Base System, *Procs. of ACM-SIGMOD* (Austin, TX, May 1978) pp. 169-180.

[FREY 87] J.C. Freytag, A Rule-Based View of Query Optimization, *Procs. of ACM-SIGMOD* (San Francisco, CA, May 1987) pp. 173-180.

[GRAE 87a] G. Graefe and D.J. DeWitt, The EXODUS Optimizer Generator, *Procs. of ACM-SIGMOD* (San Francisco, CA, May 1987) pp. 160-172.

[GRAE 87b] G. Graefe, Software Modularization with the EXODUS Optimizer Generator, *IEEE Database Engineering* **10,4** (Nov. 1987).

[HAER 78] T. Haerder, Implementing a Generalized Access Path Structure for a Relational Database System, *ACM Trans. on Database Systems* **3,3** (Sept. 1978) pp. 258-298.

[KOST 71] C.H.A. Koster, Affix Grammars, *ALGOL 68 Implementation* **Elsevier North-Holland** (J.E.L. Peck (ed.), Amsterdam, 1971) pp. 95-109.

[LEE 88] M.K. Lee, J.C. Freytag, and G.M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *IBM Research Report RJ6125* **IBM Almaden Research Center** (San Jose, CA, March 1988).

[LIND 87] B. Lindsay, J. McPherson, and H. Pirahesh, A Data Management Extension Architecture, *Procs. of ACM-SIGMOD* (San Francisco, CA, May 1987) pp. 220-226. Also available as IBM Res. Report RJ5436, San Jose, CA, Dec. 1986.

[LOHM 83] G.M. Lohman, J.C. Stoltzfus, A.N. Benson, M.D. Martin, and A.F. Cardenas, Remotely-Sensed Geophysical Databases: Experience and Implications for Generalized DBMS, *Procs. of ACM-SIGMOD* (San Jose, CA, May 1983) pp. 146-160.

[LOHM 84] G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, P.G. Selinger, Optimization of Nested Queries in a Distributed Relational Database, *Procs. of the Tenth International Conf. on Very Large Data Bases (Singapore),* **Morgan Kaufmann Publishers** (Los Altos, CA, 1984) pp. 403-415. Also available as IBM Research Report RJ4260, San Jose, CA, April 1984.

[LOHM 85] G.M. Lohman, C. Mohan, L.M. Haas, B.G. Lindsay, P.G. Selinger, P.F. Wilms, and D. Daniels, Query Processing in R*, *Query Processing in Database Systems, Springer-Verlag* (Kim, Batory, & Reiner (eds.), 1985) pp. 31-47. Also available as IBM Research Report RJ4272, San Jose, CA, April 1984.

[MACK 86] L.F. Mackert and G.M. Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, *Procs. of the Twelfth International Conference on Very Large Data Bases (Kyoto)* **Morgan Kaufmann Publishers** (Los Altos, CA, August 1986) pp. 149-159. Also available as IBM Research Report RJ5050, San Jose, CA, April 1986.

[MORR 86] K. Morris, J.D. Ullman, and A. Van Gelder, Design Overview of the NAIL! System, *Report No. STAN-CS-86-1108* **Stanford University** (Stanford, CA, May 1986).

[ROSE 87] A. Rosenthal and P. Helman, Understanding and Extending Transformation-Based Optimizers, *IEEE Database Engineering* **10,4** (Nov. 1987).

[SCHW 86] P.M. Schwarz, W. Chang, J.C. Freytag, G.M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, Extensibility in the Starburst Database System, *Procs. of the International Workshop on Object-Oriented Database Systems (Asilomar, CA),* **IEEE** (Sept. 1986).

[SELI 79] 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, *Procs. of ACM-SIGMOD* (May 1979) pp. 23-34.

[STON 86] M. Stonebraker and L. Rowe, The Design of Postgres, *Procs. of ACM-SIGMOD* (May 1986) pp. 340-355.

[ULLM 85] J.D. Ullman, Implementation of Logical Query Languages for Databases, *ACM Trans. on Database Systems* **10,3** (September 1985) pp. 289-321.

[VALD 87] P. Valduriez, Join Indices, *ACM Trans. on Database Systems* **12,2** (June 1987) pp. 219-246.

[WONG 76] E. Wong and K. Youssefi, Decomposition — a Strategy for Query Processing, *ACM Trans. on Database Systems* **1,3** (Sept. 1976) pp. 223-241.

[WONG 83] E. Wong and R. Katz, Distributing a Database for Parallelism, *Procs. of ACM-SIGMOD* (San Jose, CA, May 1983) pp. 23-29.