Compiling Object-Oriented Queries

Theodore W. Leung

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-05
Febuary 1994

Compiling Object-Oriented Queries

A Thesis Proposal

Theodore W. Leung

Department of Computer Science

Brown University

Providence, RI 02912-1910

twl@cs.brown.edu

December 6, 1993

Abstract

The effectiveness of database query optimization is dependent on the optimizer's ability to make efficient use of physical resources in a computer system. The optimizer decides how to use those resources in the plan generation step. We describe an approach to plan generation where physical level rewriting produces performance increases which cannot be obtained via source level transformations. Example optimizations serve as a demonstration of a new rule language based on list and tree pattern matching. A formalism similar to attribute grammars computes information needed by the plan generator. Many of the techniques presented are reminiscent of compiler optimizations. We conclude by describing the work that will be completed in the thesis.

1 Statement of Problem

Query Processing is one of the most desirable features of relational database systems. A user poses his query in a declarative form, and the database system devises an efficient plan for retrieving the information from secondary storage.

The answer to a query can be retrieved via a variety of plans, a query is a specification for a number of equivalent plans. At the very highest level, query optimization is a search of the space of equivalent plans specified by the query. In practice, this search is broken down into at least two steps.

The first step, source level optimization, is often referred to as rewriting. The commands of the query language are translated into operators from a query algebra. The algebra obey;s identities which are known to produce improvements in the execution of the query. The identities define a space of equivalent queries. This space is the set of all queries (in algebraic form) that can be obtained from the original query by applications of the identities.

Physical level optimization, the second step, is also known as plan generation, algorithm selection, or access path selection. During this step, the algebraic form of the query is translated into a program that actually manipulates the hardware devices where the desired data is stored. The part of the query optimizer that does this is called the plan generator. The plan generator selects an appropriate algorithm for implementing the various operators in the rewritten query, is also responsible for exploiting any special data structures, such as indexes, that could improve the performance of the query. The plan generator must search a space of equivalent plans, which it obtains by choosing between multiple algorithms and data structures for the operators in the query. Typically, a plan generator generates a new plan by devising a plan for the whole query (by starting with the rewritten query), or it enumerates the entire space of alternatives in a disciplined fashion. In many cases, a single "best" plan is selected by computing a cost metric for each alternative, and selecting the plan with the lowest cost.

The work and thesis described in the proposal focuses on the second step of optimization, physical plan generation. There has been extensive work on algorithm and data structure selection, especially in the context of relational databases. Object-oriented databases introduce a new set of problems for plan generation. A major difference between relational and object-oriented queries is that the predicates in algebra operators can be very complicated. These predicates can be queries, as in the case of nested relational queries, or they can specify object at a time navigation through the database. In either case, the pattern of data access is no longer straight forward. Control and data flow in the query bounces back and forth between the operators (which want to iterative through a single collection) and their predicates (which want to follow their path expressions thorough multiple collections). Careful management of this control and data flow can result in significant improvements in query performance.

Consider a query like "select all employees whose manager makes more than $50,000", which iterates through a collection and uses each item of the collection as the start of an object at a time path traversal. The path traversal is introduced because the query must examine the manager attribute of every employee object and the salary field of every manager object. There are many ways to evaluate this query, because there are many ways to traverse the path. One method is to read each element of the collection one at a time, and follow the path for a single element before proceeding to the next element of the collection. Under this method, we would read a single employee, then read that employee's manager, then read that manager's salary, and then begin that process anew with the next employee. Another method is to read the entire collection, follow the first level of the path for the whole collection, and repeat the process, following another level of the path for the whole collection. This would mean reading all the employees, then all the managers, and then all the salaries. These two methods of evaluating the query have a very different impact on the database buffer pool. When combined with other operators in a query one method may be much more desirable than the other. Current systems would generate a plan for one of the two methods, based on where they appeared in the search space. There is no way to directly convert from one method to another. It would be better if there was a way to move directly from one method to another when the plan generator recognized that changing methods would be beneficial. This conversion can only take place during plan generation, because the factors that distinguish the better method are only available once a plan has been generated. Since the choice of method depends on the method's interaction with the buffer usage of another operator, the conversion can only be done when the buffer behavior of the other operator is known — at plan generation time.

We argue that there are methods for improving query performance that are not concerned with algorithm selection or data structure selection, the traditional tasks of plan generators. These methods are only available during plan generation, because that is the only time when sufficient information about physical properties is available to influence the decisions of the plan generator. In the rest of this paper we propose a system where plan generation performs not only algorithm and data structure selection, but also performs transformations of the plan.

The next section reviews the current state of the art in query optimization and plan generation. After that review, we present an overview of our approach, the language formalisms that are required background, and our rule language. After the description of the rule language, Section 6 illustrates the operation of our techniques on a pair of examples. We conclude by presenting a research plan for the rest of the thesis, and a summary of our expected contributions.

2 Related Research

2.1 Plan Generation in Relational Systems

Relational plan generators search the space of relevant access structures to select the best access structure for retrieving a relation. This approach was first described in System R[76]. Ingres[80] uses a very unsophisticated plan generator which handles one variable queries only. The Ingres optimization algorithm[91] decomposes multi-variable queries into single variable queries.

2.2 Plan Generation in Object-Oriented Systems

Plan generators for object-oriented databases have continued where relational systems left off. Lanzelotte and her colleagues[59] state, “database query optimization is finding an optimal join permutation, together with the choice of an access path for each relation and an algorithm for each join and selection operator”. Relational plan generation is the choice of access path and algorithm. Some have claimed that the problem is simpler in object-oriented databases. The ObjectStore system[57, 72] selects from one of two query evaluation strategies based on whether or not an index is available. The system designers claim that the presence of paths and indexes on paths make optimization of joins much less important, but this is only true for pointer-based joins[78].

One of the points of departure for object-oriented systems has been experimentation with different plan languages and representations. This has occurred because there are additional concepts to be modeled at the physical level. Straube and Ozsu[82, 83, 81] use a “processing template” operating on streams of tuples of object identifiers to represent their plans. Each processing template represents a set of alternative plans. Unfortunately, the space of alternate plans is not very rich because the components of a processing template do not expose which algorithms are used to access data. There is a single implementation for each operator, and all variations in plans essentially come via permutations of arguments. As part of the Revelation[45, 25, 87], Bennet Vance has developed Rexall[88], a functional plan language with support for objects. Rexall allows the recognition of situations where the same physical operator is used to compute different logical operators. The two instances of the physical operator can be combined, eliminating the “algorithm” common subexpression. The TI Open OODB[12] project has also contributed in the area of plan languages by introducing a materialize operator which can be used to represent navigational object access.

Kemper, et al.[54, 55, 53] have focused on extending the notion of indexing to object-oriented systems. They have defined a new indexing structure called the access support relation, which improves the performance of path expression traversal. They have also defined some optimizations that are effective on access support relations. These include common subexpression elimination and dynamic creation of temporary access support relations. These optimizations appear to be done at the source level (before plan generation) but they are only effective if the optimizer has decided to implement a path traversal via an access support relation. Thus, these are really physical level optimizations.

There are many more opportunities for common subexpressions to appear in an object oriented query. Therefore recognition and elimination of these common subexpressions is an important optimization. Some work has been done at the source level by Cluet and Delobel[20], but they give no description of how their work impacts the plan generation stage. Lanzelotte, et al. [58, 59, 38, 60] are able to recognize common subexpressions by mapping queries onto the physical schema. Common subexpressions in path expressions appear in the class connection graph, a result of this mapping process. As previously mentioned, Kemper et al.[55] can eliminate common subexpressions that appear via access support relations, and Rexall[88] can be used to recognize "algorithmic" common subexpressions.

Lanzelotte, et al. [58, 59, 38, 60] took the first step towards rewriting at the physical level by mapping their queries onto the schema. This is done using two data structures, the class connection graph and the physical schema graph. The class connection graph is a nonnavigational representation of the implicit (path traversals) and explicit joins in the query. The physical schema graph makes the indexing and clustering relationships between actual files in the database explicit.

Once these data structures have been created, the query is optimized by generating plans from the data structures.

Both Volcano[43, 42] and Starburst[46] try to give the buffer manager hints about the way the queries will access buffer pages. Neither of them rewrites the query in order to improve its buffer usage characteristics.

2.3 Rule Based Optimization

Researchers trying to develop query optimizers for extensible database systems adopted rewrite rules as a mechanism for extending the set of available optimizations. Typically, rules have a left hand side, which is used to match a pattern in the query, and a right hand side, which specifies a replacement for the matching pattern. The rules are processed by a rule engine, which is controlled by a search strategy. Rules are applied to generate new versions of the query, and the search strategy determines which rule will be applied next. In some systems the search strategy also determines which versions of the query will have the rule applied to them. Usually, rules are applied to rewrite the query in algebraic (source) form, and then a set of rules that maps algebra operators to physical operators is used to generate a plan. The key components in a rule based optimizer are the language for describing rules, the search strategy, and the rules themselves.

Rule based optimization was first suggested by Freytag[34, 36, 35]. His initial proposal calls for two layers, where the first layer rewrites the Lisp-like intermediate representation of the query. The rewriting phase does not distinguish between source and physical rules, although it possesses both. The second layer translates the intermediate representation into an iterative program.

The next advance in rule based optimization came with the Exodus optimizer generator[41, 39]. This system allows a database system implementor to generate a query optimizer from a set of grammar like rewrite rules. The rule language is grammar like, with the addition that rules may have an associated condition that determines whether or not the rule is applicable. Conditions are written in C, and the database system implementor has the option of dropping back into C when the rule language is not expressive enough. The generator allows the implementor to specify cost functions, and uses a hill climbing search strategy to determine which rule to apply next. Rules are classified as transformation (source to source) or implementation (source to physical) rules, but no physical to physical rules are used. The rule language does not allow the predicate to be manipulated.

The Volcano optimizer generator[43, 42] is the successor to the Exodus optimizer generator[41]. The major features introduced in Volcano are improvement of the search strategy, implementation of cost as an abstract data type, ability to memoize the results of previous optimizations, and the use of enforcers and physical property vectors to force the plan generator to make particular choices. Logical and physical expressions are now clearly distinguished in the representation. The TI Open OODB project[12] has implemented its query optimizer with the Volcano optimizer generator. One lesson learned from their experience is that it is desirable to be able to manipulate the predicate part of an operator, but they found this hard to do in the Volcano optimizer generator.

The Starburst[46, 47, 73, 64, 61] extensible database system separates query optimization into two levels, query rewrite and plan optimization. The system uses a different rule language for describing the processing at each level. During query rewrite, the query graph model (QGM) representation of the query is rewritten into an equivalent QGM query. Their rule language for this phase is C. The plan optimization phase uses grammar-like rules, called STARS, to construct plans from the bottom up. Alternative plans are generated simultaneously eliminating the need for any kind of rewriting. STARS also have required and available properties, which correspond very closely to the notion of inherited and synthesized attributes in attribute grammars. The attribute grammar evaluation mechanism is simplified tremendously because there are only two attributes.

Fegaras, et al.[31] have demonstrated a query optimizer generator which uses compile-time reflection in ML. Rewriting is controlled via a rule-based term rewriting system that has the flavor of attribute grammar evaluation. The attributes from the term rewriting system are used for context control. The rules are very succinct because the full power of ML, including pattern matching, is available due to the specification of rules via compile-time reflection. This makes it much easier to write the necessary support functions. They note that they do not add functionality relative to the Volcano optimizer generator, but provide conciseness and clarity in their rules. The use of the attribute grammar for context control is interesting. Our work uses a tree query language for context control, and an attribute grammar like system for propagating analysis information. Also, ML's pattern matching is not as expressive as our tree query language.

2.4 Other Extensible optimizers

Extensible optimizers which are not based on rules have also been proposed. The EPOQ optimizer[69, 70] modularizes the set of optimizations available to the optimizer. These modules are called regions, with each region providing a specific optimization service to the optimizer. A region is viewed as a black box which responds to a protocol and which operates on an annotated common representation for queries. No other assumptions are made about about the internal structure or operation of regions. This allows great freedom in the actual implementation of regions, allowing individual optimizations to be expressed in whichever formalism is most appropriate. The overall control and coordination of the regions is the responsibility of the optimizer. Plan generation fits into the framework as a region, but no specifics are given regarding plan generation.

Kemper, et al.[56] discuss an architecture for an extensible optimizer which is based on the blackboard paradigm for artificial intelligence. The unit of extensibility is the knowledge source –

each knowledge source is an expert at a particular kind of optimization. The optimizer produces query evaluation plans that are DAG's of algebraic operator applications. There is no lower level plan generation in the sense that we have been discussing.

2.5 Analogy to Compilers

There are some interesting analogies between techniques developed by compiler researchers and the techniques that we have developed.

Compiler researchers have used tree pattern matching to help build portable and efficient code generators. The Twig system [4] uses a tree matching language and a dynamic programming algorithm to generate code generators. Their tree language is fairly simple in comparison to our tree query language, but it allows them to specify code generation and improvement in terms of tree creation and tree transformation. Code generation systems based on Bottom Up Rewrite Systems (BURS), like IBURG[33] also use tree pattern matching, but restrict its expressiveness in the interest of efficiency.

A little work has been done to try to manage virtual memory[2] or processor cache memory[15, 16] at code generation time. The work on virtual memory management was done in the context of FORTRAN, and concentrated on distributing the control of multiple loops and reordering loop indices when performing array operations. The reported speedups are close to an order of magnitude. Compiler directed cache management views the cache as an extension of the register set and attempts to apply generalized register management techniques to manage the cache.

In the database world, Lieuwen[63] has found that performing compiler style analyses on database programming language loops allows those loops to be turned into joins. Cluet & Moerkotte [21], in their work on nested queries have found dependency information to be very useful.

These parallels make us confident that our techniques will prove useful.

2.6 Buffer Management

Buffer management has always been a factor that affects the performance of database systems[29]. Most work on buffer management and query optimization has focused on finding ways to give the buffer manager hints about the way that buffer pages will be used[74, 75, 17, 71, 30]. These hints are used to control buffer page replacement. This kind of hinting has also been applied to indexing structures[13]. Cornell and Yu[24] integrate buffer management with the query optimization process including by buffer residency and contention for buffers as a part of the optimization problem. Their techniques were developed in the context of the relational model and focus primarily on improving the performance of joins.

2.7 OODB Access Structures

2.7.1 Assembly

Object assembly[51, 52] is an operator which is used to retrieve objects in a set-oriented manner — that is — other than one at a time. The assembly operator is able to do this by looking at the actual storage layout of objects on the disk, even if the query invoking assembly asks for the objects in another order. This is both an advantage and disadvantage, since the assembly operator could "outsmart" the query optimizer but actually end up performing a pessimization. This can happen when the query optimizer is trying to order the pieces of the query in order to control buffer usage. The Revelation and Volcano optimizers do not attempt to perform such ordering.

2.7.2 Indexing

Indexing has been a major data structure used to improve database query performance[22, 67]. Indices make use of a search data structure like b-tree to reduce the amount of data that must be searched in order to find the answer to a query. There have been a number of proposals for indices in object-oriented databases[68, 10, 9, 93, 54, 55], each with their own strengths and weaknesses. The ability to make use of indices is a crucial requirement for a plan generation system.

2.7.3 Clustering

The objective of clustering is place related objects close together physically, so that retrieving one object from disk causes all the related objects to be retrieved. There have been many proposals for single and multi-page clusters [14, 11, 77], as well as performance studies of the impact of clustering[49]. There has also been extensive work on methods for deciding which objects to place in a cluster[8, 85, 86]. Support for clustering is essential for good performance in object-oriented systems.

3 Approach

Ultimately, the query must be evaluated by using specific algorithms to access specific data structures on secondary storage. In order to achieve maximum performance, an optimizer must make sure that it is using the best algorithms on the most efficient data structures, and that resources like disk arms and buffer pools are used efficiently.

In most current systems, a query is parsed into an intermediate form, and then rewritten using source to source rules. After the source to source rules have been exhausted, source to physical rules are used to map the rewritten query onto structures that can be executed by a query execution engine. The source to physical rules are guided by information from the database catalogs. Some of that information can be used to compute cost measures. This process is shown in figure 1. The rule languages for the source to source and source to physical rules are usually the same[41, 42],

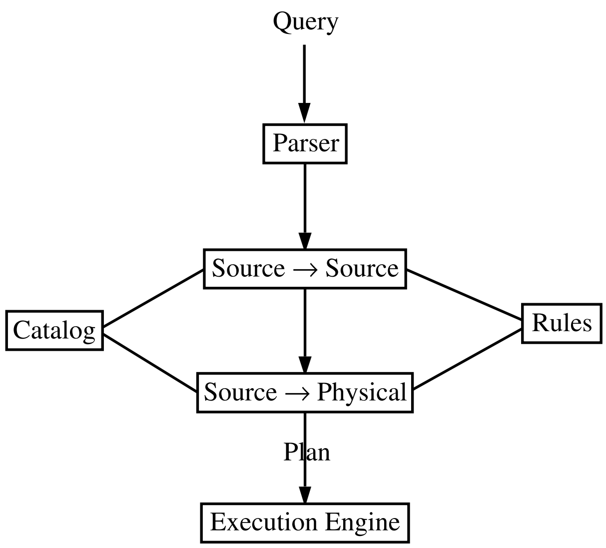

Figure 1: Traditional Optimizer Architecture

with Starburst[47] being a notable exception. Each rule language requires its own rule evaluation engine.

Our approach to plan generation extends the process and adds an additional step to this process. Since our work is a part of the EPOQ[69] project, the source to source transformation is performed by regions. Some of the regions may be implemented in a rule-based fashion, but others may not be. The additional step is that we use a new kind of rule to rewrite the physical plan form of the query after the source to physical rules have mapped the query from the logical algebra to the plan language. This allows the plan generator to use optimizations that can only be performed at the physical level. The physical to physical rules make use of information from the catalog, and an attribute grammar system is used to annotate the query with analysis information. That analysis information helps to guide the optimization process. Our modified process is illustrated in figure 2. The same rule language is used for the source to physical and physical to physical rules. This rule language needs its own rule evaluation engine, since it is distinct from whatever rule languages are defined by the regions performing source to source transformations. There is also the possibility

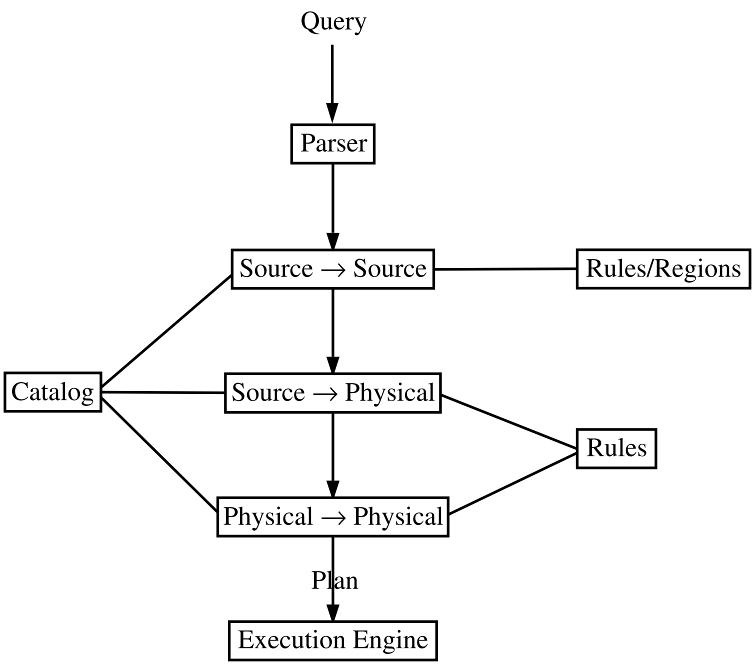

Figure 2: Our Optimizer Architecture

for feedback between the physical and source level optimizers, but we do not account for this at the moment.

In order to implement this approach we have developed a rule language for writing physical level rules. This language incorporates a query language on trees, allowing us to to perform complex pattern matching on the physical representation of queries. It also allows us to use non-local information in the query during the optimization process. Non-local information is information that extends beyond the "scope" of a single pattern. We also have a plan language that allows us to demonstrate the benefits of physical level optimization. Attribute grammars form the basis of a system for annotating queries with non local information.

Algorithmically, the plan generation process is:

LOOP until < plan is "good" >

1. Use the attribute grammar formalism to compute any additional or new information from

physical level information.

2. Use a rule language based on the AQUA query language to translate the current plan into a better plan.

There are two classes of optimizations that can only be performed at the physical level. The first class requires information or a representation that is only available at the physical level. An example of this kind of information would be the order in which a path expression will be traversed, or the buffer behavior of a particular operator. The other class of optimizations generate or manipulate physical structures. A dynamically created access structure is a physical structure that might be created by a physical level optimization.

This new methodology is not restricted by the choice of logical algebra or physical plan language. The rule language and the attribute grammar framework will remain the same for any choice of algebra and plan language. The rules would need to be updated or rewritten in order to support a new algebra or plan language. This is the same for any rule-based extensible optimizer. The more expressive the plan language is, the greater the opportunities for plan level optimizations, so changing the plan language could result in reduced opportunities for physical level optimization.

4 Language Formalisms

This section describes some of the language formalisms that we have developed for use in this work. They serve as background for introducing the rule language and our examples. The AQUA query algebra is the input language for our query optimizer. We have developed a plan language in order to illustrate some of the kinds of physical optimizations that are possible. The AQUA tree query language contributes a large part of the functionality for the rule language.

4.1 AQUA Query Algebra

This research occurs within the context of the EREQ project at Brown, Oregon Graduate Institute of Science and Technology, and the University of Wisconsin. The EREQ project has developed an object-oriented query algebra, AQUA, [62], which is the input language for the EPOQ [69] optimizer being developed at Brown. The output of the source level optimization component of EPOQ is the input to our plan generation system. In this section, we present the parts of AQUA necessary to understand the examples presented in the rest of the paper.

Every AQUA object has a type, and presents an interface which is a set of function signatures. The functions in the interface (called *attributes*) specify the externally visible behavior of an object. The attributes may return data from the representation of the object or may return data that is computed from the representation of the object. Syntactically a *path expression* is a variable name or object name followed by sequence of attributes. The attributes are separated from the variable name and from each other by "."'s. The variable at the beginning of a path expression is called the *head* of the path-expression, while the last attribute in the path expression is called the *tail*. The *head-type* is the type of the head of a path-expression, and the *tail-type* is the type of the tail (actually the return type of the attribute in tail position). A path expression represents a series of *paths* through the database. A *path* begins at an object which has the type of the head of the path, and includes the sequence of objects that is found by beginning with the object at the head and following the attribute names in the path expression. The act of reading all the objects in a path denoted by a path expression is called *traversing* the path expression. The *head-object* is the object at the head of a path, and the *tail-object* is the object at the tail of the path. A *subpath* is a subsequence of a path.

The AQUA algebra defines algebraic query operators for a number of bulk types, including

Sets, Multisets, Lists and Trees. The basic syntax of all algebra operators is opname(<predicate or type/equality parameters>)(input parameters). The set algebra includes the familiar **select** and **apply** (map) operators. The algebra includes a treatment of user defined equality. Predicates can invoke methods on elements of a set. The **choose** operator nondeterministically selects one element from a set.

The EPOQ optimizer delivers the query to the plan generator in an intermediate form, the EPOQ query rep (hereafter query rep) [69, 66]. This representation is used to represent the query both in source form, and in physical plan form. The query rep is a tree-like data structure, where operators or functions are represented as function nodes which take data nodes as their inputs and produce data nodes as their outputs. This creates a tree with alternating layers of function and data nodes.

4.2 Plan Language

We need a language to describe plans. The language should make evaluation order explicit because physical level optimizations may rely on the history of operation performed up to a certain point. This also allows a plan generator to reason about data that may still reside in the database buffer pool. The plan generator may elect to rearrange the order of computations in order to leave beneficial data in the buffer pool. We also want a language which makes buffer pool allocation and access explicit, so that the plan generator can adjust the allocation of buffer pages or alter the order of buffer access.

Our plan language is a language based on functions (everything is accomplished by function calls), but is not functional, since some of the functions produce side effects. We present the interesting features of the language before listing the rest of the operators. The **partition** operator takes an expression representing a set, and divides it into partitions occupying a certain number of pages (the specific number is an argument to **partition**). This operator is used to control the unit of transfer between operators. We express path expressions as a series of nested **apply** operators, where each **apply** applies a single element of the path expression. This representation of path expressions, along with the **partition** operator, allows the plan generator to generate plans that perform depth first, breadth first, or mixed depth or breadth first path expression traversal. All operators that generate results generate them in a lazy manner (i.e. as they are needed), making pipelining a default feature of the plan language.

The operators in the plan language are:

partition(Q, int) - divides the set Q into partitions of size int pages.

filter(Q,F) - returns all the elements of Q that satisfy the boolean function F.

apply(Q,F) - applies the function F to each element of Q.

indexFilter(Q,N,F) - does the same thing as **filter**, but uses the index named N to help it find matching elements.

hashFilter(Q,N,F) - does the same thing as **filter** but uses the hashtable named N to help it find matching elements.

join(Q,Q,F) - implements an AQUA **tup_join** (tuple-join) using F as the join predicate.

loopJoin(Q,Q,F) - does what **join** does, but uses the nested loops method for evaluating the join. The first Q argument is the outer loop and the second Q argument is the inner loop.

mergeJoin(Q,Q,F) - does what **join** does but uses the merge join method for evaluating the join.

hashJoin(Q,Q,N,F) - does what **join** does but uses the hash table named N on the second Q argument to evaluate the join.

flatten(Q) - takes a set of sets and returns the set containing the union of all the nested sets in its input.

incrementallyBuildPathIndex(N,Q,P) - traverses the path P and incrementally adds an entry to the dynamically created index named N, that links the head-object with the tail-object. The path is assumed to start from the query Q.

incrementallyBuildCluster(N,Q,P) - traverses the path and incrementally adds an entry to the dynamically created cluster named N that links the head-object with the tail-object. The path is assumed to start from the query Q. The database is reorganized in order to create the cluster.

readPathIndex(N,Q,P) - uses the path index named N to traverse a previously memoized shared subpath,P. The starting point for the path index comes from Q.

readCluster(N,Q,P) - uses the cluster named N to traverse a previously memoized shared subpath, P. The starting point for the cluster comes from Q.

The grammar for the plan language is:

Q ::= partition ⟨(⟩ Q ⟨)⟩ int | ⟨filter|apply⟩ ⟨(⟩ Q ⟨,⟩ F ⟨)⟩ |
⟨indexFilter|hashFilter⟩ ⟨(⟩ Q ⟨,⟩ N ⟨,⟩ F ⟨)⟩ |
⟨join|loopJoin|mergeJoin|⟩ ⟨(⟩ Q ⟨,⟩ Q ⟨,⟩ F ⟨)⟩ |
⟨flatten⟩ ⟨(⟩ Q ⟨)⟩ |
⟨incrementallyBuildPathIndex|incrementallyBuildcluster⟩ ⟨(⟩ N ⟨,⟩ Q ⟨,⟩ TP ⟨)⟩ |
⟨readPathIndex|readCluster⟩ ⟨(⟩ N ⟨,⟩ Q ⟨)⟩ | N ■

AL ::= var | AL ⟨,⟩ var ■

PL ::= ⟨(⟩ AL ⟨)⟩ ■

$\mathrm{F} ::= \langle\lambda\rangle\ \mathrm{PL\ Q}\ \mid \langle\lambda\rangle\ \mathrm{PL\ P}\ \mid \langle\lambda\rangle\ \mathrm{PL\ E}\ \blacksquare$

$\mathrm{E} ::= \mathrm{E}\ \langle > \mid < \mid = \mid != \mid >= \mid <= \rangle\ \mathrm{P}\ \mid \mathrm{N}\ \mid \mathrm{P}\ \blacksquare$

$\mathrm{N} ::= \text{identifier}\ \blacksquare$

$\mathrm{TP} ::= \text{identifier}\ \mid \mathrm{TP}\ \langle . \rangle\ \text{identifier}\ \blacksquare$

$\mathrm{P} ::= \text{var}\ \langle . \rangle\ \mathrm{TP}\ \blacksquare$

Initially we tried a lower level language, which looked more like an assembly language. The advantage of that language was that it allowed very direct control over buffer management issues and order of evaluation. The disadvantage was that doing pattern matching was too difficult. The current language is a compromise. The partition operator allows some but not all of the control allowed by the assembly language, but the current quasi-functional language makes rule pattern matching more reasonable. It should be easy to adapt other plan language level proposals like dynamic query plans [44] and object assembly [51] fit into this framework. We have our own plans to add what we call stylized plan operators (see Sec. 7) to the language as part of our future work.

4.3 Tree Query Language

Following the trend in recent optimization research, we want to define a rule based language for expressing optimizations. The benefits of rule based optimizers are well known. They provide an extensible and declarative framework for specifying optimizers. Our attempt to perform complicated optimizations at the physical level induces some additional requirements which are not satisfied by existing rule based languages. In particular, we need the ability to look at large portions of the trees (since the query rep is a tree), and the patterns which describe when a rule matches need to be able to take into account more of the surrounding context of the tree.

As part of the EREQ project, a query language for trees and lists has been developed [84, 89]. Our rule language adapts and extends that work, allowing us to use the query language to specify pattern matches in the tree representation of queries, and also allows us to use queries in computing the actions to be taken when a rule is to fire.

The tree query language consists of a set of query operators over ordered trees of arbitrary but finite arity, and a predicate language that specifies patterns over those trees. The definition of subtree and subgraph in this language are a little startling. A *subtree* rooted at a node must contain all the children of that node, for the children of those child nodes, if any child is included, all the children must be included. A *subgraph* rooted at a node is some subgraph of the graph obtained when the tree is viewed as a graph.

For our purposes, we assume the availability of a set of predicates which operate on tree nodes and which return true or false. This set of predicates includes a set of predicates whose names are the names of logical or physical query operators. The predicate returns true if the node being matched is a function node representing the operator with the same name as the predicate. For example, the predicate "select" will return true if the node being matched is a function node representing a **select** operator. At the simplest, a tree pattern is represented by a predicate for the root node of the pattern, followed by a tree pattern for each of the children:

tree-pattern ::= tree-predicate ⟨(⟩ child-pattern-list ⟨)⟩ ■

tree-predicate ::= predicate-name ■

child-pattern-list ::= tree-pattern | child-pattern-list ⟨,⟩tree-pattern ■

child-pattern ::= tree-pattern ■

In addition to the set of predicates described above, the predicate also includes a set of metacharacters. The metacharacter "?" is shorthand for the union of all predicate names – it matches anything. So that the full definition of tree-predicate is

tree-predicate ::= predicate-name | ⟨?⟩ | tree-predicate ⟨|⟩ tree-predicate ■

There are a few twists on what constitutes a tree pattern for a child of a node. A child tree pattern can be a regular tree pattern. It can also be the metacharacter "?*",which matches any number of "?" matches. This allows the pattern language to work with trees of unknown arity. The tree language is derived from regular expressions — tree patterns can be combined using intersection '|', concatenation $\circ$, or Kleene closure *. Kleene closure can be viewed as self-concatenation. In the case of strings or lists, it is obvious what self-concatenation means. For trees, there is uncertainty about where the new copy of the pattern should be concatenated to the old one. The notion of a concatenation point is introduced to remove this uncertainty. The algebra provides three concatenation points, α, β, and γ, so that multiple self-concatenations can be performed. The introduction of multiple concatenation points also necessitates multiple versions of the Kleene star, subscripted with the appropriate concatenation point. To self-concatenate with the β concatenation point, one writes T^{*_β}. The real definitions of tree-pattern and child-pattern follow below: ($*_\alpha$ has been use to represent the *'s for β and γ.)

tree-pattern ::= tree-predicate $\langle (\rangle$ child-pattern-list $\langle) \rangle$ |
 tree-predicate $\langle (\rangle$ child-pattern-list $\langle)^{*_\alpha} \rangle$ | tree-pattern $\langle \circ \rangle$ tree-pattern ∎

child-pattern ::= tree-pattern | $\langle ?^* \rangle$ | $\langle \alpha \rangle$ | $\langle \beta \rangle$ | $\langle \gamma \rangle$ | $\langle$tree-pattern$\rangle^{*_\alpha}$ ∎

This pattern language allows queries to specify trees or pieces of trees that satisfy a particular pattern. This is used to identify relevant parts of our query representation, which is a tree. The tree pattern language can also appear in the predicate of a set operator if the operator is working over a set of trees.

The query operators over trees are described in detail in appendix A. The grammar for tree queries follows below:

tree-query ::= $\langle$select$\rangle$ $\langle (\rangle$ predicate $\langle) \rangle$ $\langle (\rangle$ tree $\langle) \rangle$ |
 $\langle$sub_select|all_anc|all_desc$\rangle$ $\langle (\rangle$ tree-pattern $\langle) \rangle$ $\langle (\rangle$ tree $\langle) \rangle$ |

$\langle$PT|PSG|root|leaves|nodes$\rangle$ $\langle()$ tree $()\rangle$ |
$\langle$im_ancestor|im_descendant$\rangle$ $\langle()$ node $()\rangle$ $\langle()$ tree $()\rangle$ |
$\langle$tree$\rangle$ $\langle()$ node $()\rangle$ ■

As an example, the query that retrieves all subgraphs of Q that represent a join whose left input contains a select somewhere in it is **sub_select**($join$ $((?\ (\alpha\ ?)|?\ (?\ \alpha))^{*\alpha}\ select)\ ?)(Q)$ The **sub_select** operator selects all subgraphs of Q that match the tree predicate. The predicate expression $(?\ (\alpha\ ?)|?\ (?\ \alpha))^{*\alpha}$ allows the select to appear as either the left or right child or the left join input. The select after that is the base case for the induction implied by the $^{*\alpha}$.

4.4 Attribute Grammar

Our rule language provides a powerful but succinct notation for specifying transformations to be performed on physical versions of queries. In order to make physical level optimization effective, we also need to have good analysis information so we can make good decisions about when to apply transformations. This analysis information is also useful for helping individual transformation rules make internal decisions. For example, the rule (in Sec 6) for example 2's optimization is rather naive. The paths P and P1 are selected (nondeterministically) randomly. Yet the order in which the paths appear in the query can affect the effectiveness of the memoization function. If information about the dependency relationships between the two path expressions was available, it could be used to make a better decision about whether a path should be P or P1.

In compilers[5, 32], attribute grammars are used to declaratively compute and propagate information up and down abstract syntax trees. This information is then available for use during optimization and code generation. We adapt attribute grammars to our own situation by presenting a version of an attribute grammar formalism that can be used to compute analysis information for queries, both in source and physical forms.

attribute-grammar ::= grammar-rule | attribute-grammar grammar-rule ■

grammar-rule ::= non-terminal ⟨{⟩ inherited-attribute-eqn ⟨}⟩ ⟨::=⟩
rhs ⟨{⟩ synthesized-attribute-eqn ⟨}⟩ ⟨.⟩ ■

inherited-attribute-eqn ::= attribute-eqns ■

synthesized-attribute-eqn ::= attribute-eqns ■

attribute-eqns ::= attribute-eqn | attribute-eqns ⟨;⟩ attribute-eqn ■

attribute-eqn ::= identifier ⟨.⟩ identifier ⟨=⟩ attribute-expr ■

attribute-expr ::= identifier | identifier ⟨.⟩ identifier |
attribute-expr op attribute-expr | attribute-expr op attribute-literal ■

attribute-literal ::= literal ■

rhs ::= non-terminal | terminal | rhs non-terminal | rhs terminal ■

non-terminal ::= identifier ■

terminal ::= identifier ■

Figure 3: The syntax of Attribute Grammars

Appendix B contains a full attribute grammar that computes attributes for simple dependency information. A query tree that has been annotated with simple dependency information is shown in figure 4. The boxes next to the node names indicate the annotations for the nodes. The abbreviations DD and ID stand for DirectlyDependent and IndirectlyDependent as specified in the appendix.

Dependency information is not the only information that can be computed using attribute grammars. We can use the formalism to propagate cost information and physical properties up and down the query. A more sophisticated set of dependencies is also necessary.

Beyond the issue of additional useful attributes, is the issue of when the attributes are actually computed. At present, the logical level grammar is computed when the translation from source level to physical level is performed. The physical grammar is a little more problematic. We could recompute the grammar after each rule is applied, but this is very expensive, and may often be wasteful. Application of incremental attribute grammar techniques would be useful here. Another direction to be pursued is to use feedback between the rule language and the attribute grammar system to cause the attribute grammar to be recomputed when it is necessary.

5 Rules

We are now ready to describe our language for writing rules. It is a simple condition action language, where the truth value of the condition determines whether or not the action is performed. The language must be rich enough to match meaningful patterns in the physical representation of queries, and must provide enough action primitives to perform useful transformations. This means that the rule language must be able to find patterns in trees and it must be able to operate on trees, creating new fragments of trees, or deleting or replacing fragments of trees.

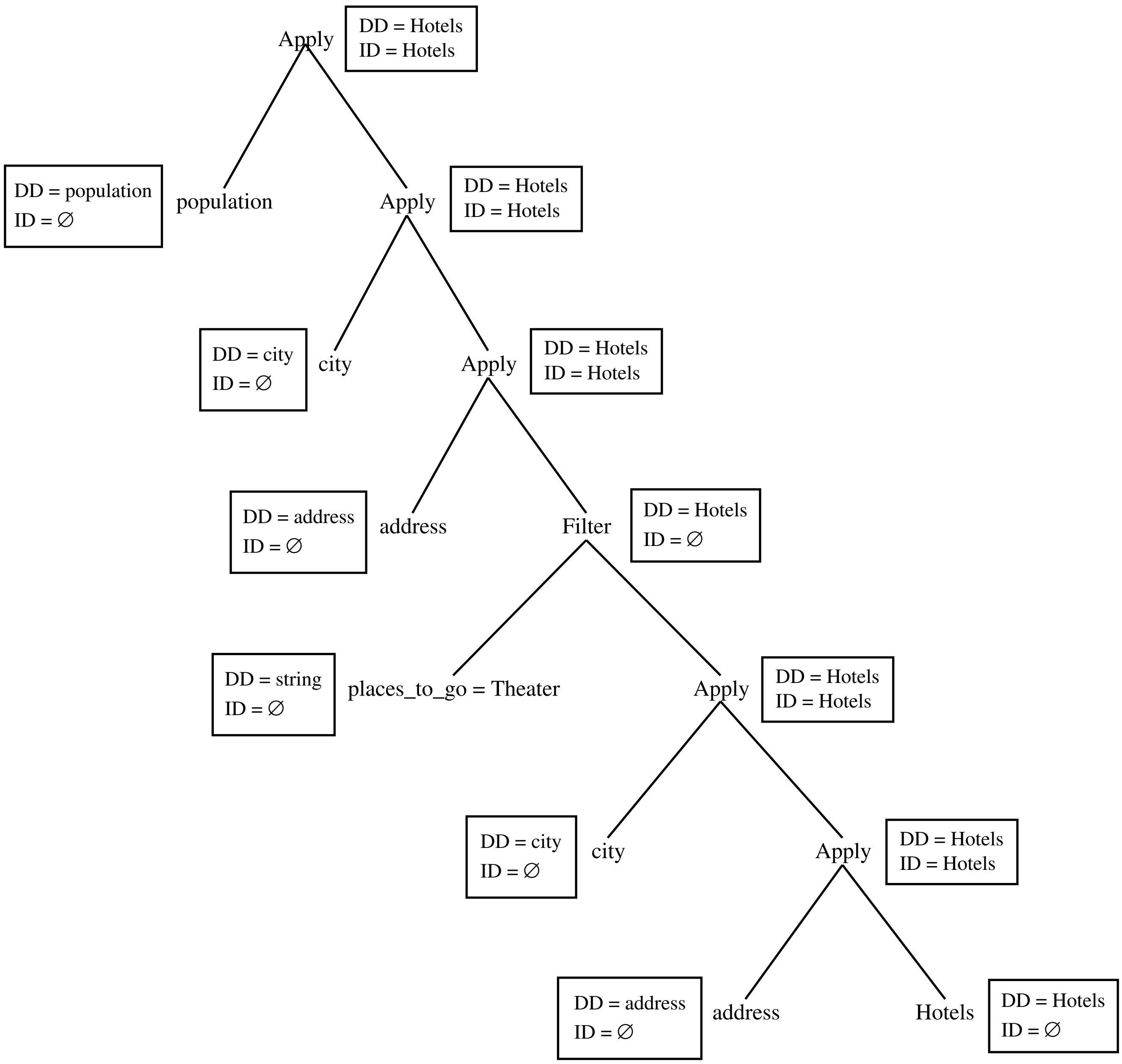

Figure 4: Query with Dependency Annotations

5.1 Syntax

Syntactically, rule language is fairly standard. Rules take the form:

rule ::= ⟨if⟩ condition ⟨then⟩ ⟨let⟩ decl ⟨in⟩ actions | ⟨if⟩ condition ⟨then⟩ actions ■

condition ::= boolean-expr ■

actions ::= action | actions ⟨;⟩ action ■

decls ::= decl | decls ⟨;⟩ decl | ϵ ■

decl ::= var ⟨:=⟩ expr ■

expr ::= tree-query | | ■

action ::= expr | ⟨REPLACE⟩ tree ⟨WITH⟩ tree-expr | ⟨CREATE⟩ tree-expr
tree ::= tree-expr ■

expr ::= tree-query | tree-pattern | expr | ■

predicate ::= lambda-expr ■

lambda-expr ::= ⟨λ⟩ ⟨(⟩ parameter-list ⟨)⟩ expr ■

parameter-list ::= name | parameter-list ⟨,⟩ name ■

boolean-expr ::= expr ⟨AND⟩ expr | expr ⟨OR⟩ expr | expr ⟨=⟩ expr |
expr ⟨!=⟩ expr | expr ⟨>⟩ expr | expr ⟨<⟩ expr | expr ⟨>=⟩ expr |
expr ⟨<=⟩ expr ■

expr ::= primary-expr | binary-expr |
primary-expr ::= ⟨(⟩ expr ⟨)⟩ | var | literal | tree-query |
function-name ⟨(⟩ expr-list ⟨)⟩ ■

binary-expr ::= expr ⟨in⟩ expr | expr binary-op expr ■

expr-list ::= expr | expr-list ⟨,⟩ expr ■

var ::= ⟨$⟩ name | ⟨$⟩ name ⟨:⟩ name ■

literal ::= ⟨⟩ | numeric-constant | ⟨”⟩ string-constant ⟨”⟩ ■

tree-expr-list ::= tree-expr | tree-expr-list ⟨,⟩ tree-expr ■

tree-expr ::= var | plan-operator ⟨(⟩ tree-expr-list ⟨)⟩ ■

A communication path is provided between the pattern matching language and the rule language. This path allows any tree predicate to be given a name, so that whatever piece of tree matches the name can be manipulated by the rule language. This is accomplished by modifying the definition of tree-predicate slightly.

tree-predicate ::= predicate-name | $\langle\$\rangle$ name $\langle:\rangle$ predicate-name | $\langle ? \rangle$ |
$\langle\$\rangle$ name $\langle:\rangle$ $\langle ? \rangle$ | tree-predicate $\langle | \rangle$ tree-predicate ■

Any predicate or "?" metacharacter can be named. That name can then be used to refer to the piece of tree that matched the predicate or metacharacter. The distinguished variable named Q always denotes the query being operated on.

5.2 Semantics

The basic semantics of rule evaluation is simple. The rule engine evaluates the conditions of all the rules to determine which ones can execute actions. If only one rule matches, then its actions are performed. If more than one rule matches, a disambiguation policy determines which rule is selected. The choice of a good policy is part of the research plan (Sec. 7). Beyond that, the only "working memory"[23] available to the rule engine is the query being transformed and the annotations that have been made to it. All locally defined variables are local to a rule invocation and do not persist after a rule has executed. If it becomes useful to add a notion of global working memory, this can be done.

5.3 Issues

The fact that we are using a query language for pattern matching means that there is the possibility for multiple matches in the condition part of rules. It is not clear what should be done in situations where a query returns a non-singleton answer. There are at least two reasonable alternatives: 1) select one of the multiple matches and proceed, or 2) have a set-oriented semantics for rules so that the action part of the rule applies to each match in the condition part. The final semantics for this case has not been resolved because we do not yet have enough experience writing rules in this language.

Since we are performing optimizations on the physical plan versions of the query, we must be aware that the same logical level notion may have multiple representations at the physical level. As an example, path expressions are represented by nested **apply** operators, but the exact form can vary widely, depending on whether a depth or breadth first strategy has been adopted for traversal, or on whether **partition**ing has been performed or not. This implies that rules for a particular optimization must be able to operate on the range of representations for the particular portion of the query which is of interest. There are a number of solutions which we are considering at the moment:

1. Have multiple sets of rules for each optimization, with one rule for each representational variation

2. Use some kind of template based scheme to generate the set of rules necessary to cope with representational variations

3. Convert all representational variations to a common form and track the relationship between the common form and the original variant. This means that optimization rules must describe how they affect the relationship between the common form and the variant.

There is also a decision about when to convert the query from source to physical forms. At the present, we assume that the query will only be translated from source to physical form once, and that all subsequent optimizations will occur via rewrites from the original physical form. An

class Hotel (name : string, address : Address, stars : integer)	class Address (country : Country, city : City, street : string, number : integer)	class City (name : string, population : integer, places_to_go : string)

Figure 5: Schema for the Examples

alternative strategy is to allow the translation from source to physical forms to happen as often as deemed necessary by the plan generator. This is an interesting topic for future work.

6 Example Optimization

We will be using a pair of examples to demonstrate the use of our techniques and results. In some cases, the examples or the solutions to the examples have been simplified for the purpose of explaining the concepts being developed in the thesis.

The example queries operate over the schema in figure 5:

6.1 Example 1

The first query looks for all Hotels that are in Cities that have famous Theaters as tourist attractions. This is how the query would be written in AQUA:

$\mathbf{select}(\lambda(h)\, h.address.city.places\_to\_go = Theater)(Hotels)$

This query is an instance of a common class of queries, which iterate through a collection, typically via **select** or **apply**, applying a path expression to each element of the collection. Most object-oriented databases evaluate such queries by iterating sequentially through the collection, fully applying the path expression to each element of the collection before proceeding to the next element. There are two kinds of optimizations which are used to improve the performance of this class of queries. One uses some kind of indexing structure to associate the values at the end of the path expression with the values at the beginning. The other makes use of clustering to put objects which are traversed via path associations on the same disk page. Clustering is generally only effective for one path expression, since it requires physically placing the objects involved on a disk page. For clustering to be useful for more than one path expression, the database must be able to handle multiple copies of the same object on disk. This is generally undesirable since it creates difficulty for updates. If no index or clustering structure exists, then sequential search is the only evaluation method available. One of our goals is to provide alternatives to the strategy which walks to the end of each path for each object in the collection. We will call this strategy depth first traversal. In depth first traversal, the disk arm must jump all over the disk in order to read individual pages for each of the elements of the path being traversed. This situation does not occur in relational databases since there is no notion of navigational access. This kind of access creates performance difficulties as the iterative process (**select**) in the query and the navigational process in the query (path expression traversal) compete for control of the disk arm and for resources in the buffer pool. This competition means that there are times when depth first traversal is not the best evaluation strategy. Depending on the situation, breadth first traversal, in which all the elements for each element of the path expression (beginning at the root) are read in before proceeding to the next element of the path expression, can be a better choice. There are also intermediate positions between the extremes of depth-first and breadth first traversal. In order to generate efficient plans for this class of queries we would like the ability to control which retrieval strategy is used. In addition, the number of physical units actually transferred can have a large effect on the contents of the buffer pool. We would like to be able to exercise control over the number of physical units transferred during the traversal of the individual paths that must be traversed. We will show an optimization that allows us control path expression traversal order and to determine how many physical units will be transferred at each step of a traversal. This control is accomplished by representing a path expression as a series of nested **apply** operations (see Sec 4.2), and by using **partition** to divide sets into units of smaller size that can be retrieved from the disk efficiently. A plan language formulation for the query in Example 1 is:

$$\textbf{Filter}(\textbf{Apply}(\textbf{Apply}(Hotels, \lambda(h)\, h.address), \lambda(a)\, a.city), \lambda(c)\, c.places\_to\_go = Theater)$$

(The source to plan translations in the two examples are not the best translations of the source expressions into the plan language. For the sake of brevity, the rules used to perform the translation have been omitted, but they are easily expressible using our rule language (see Sec 4.3).)

This plan causes the path expression to be traversed in breadth-first order, since *address* will be applied to all the Hotels before *city* is applied to any of the addresses. We want to add **partition** operators to force the path expression to be read in depth-first order. This is accomplished by writing the plan like this:

$$
\begin{array}{l}
\textbf{Flatten}(\\
\quad \textbf{Apply}(\\
\quad\quad \textbf{Partition}(\\
\quad\quad\quad \textbf{Flatten}(\\
\quad\quad\quad\quad \textbf{Apply}(\\
\quad\quad\quad\quad\quad \textbf{Partition}(\\
\quad\quad\quad\quad\quad\quad \textbf{Flatten}(\textbf{Apply}(\textbf{Partition}(Hotels, 1), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda(sh)\, \textbf{Apply}(sh, \lambda(h)\, h.address))), \\
\quad\quad\quad\quad\quad\quad 1), \\
\quad\quad\quad\quad\quad \lambda(sa)\, \textbf{Apply}(sa, \lambda(a)\, a.city))), \\
\quad\quad\quad 1) \\
\quad\quad \lambda(sc)\, \textbf{Filter}(sc, \lambda(c)\, c.places\_to\_go = Theater)))
\end{array}
$$

Adjust the integer parameter to **partition** allows us to control how the path expression is evaluated. We now begin the development of a rule that looks for a place in the path expression where the arguments for two "adjacent" **partition** operators are not the same. The argument for the outer **partition** is changed to match the argument for the inner **partition**. This reflects the fact that it is only sensible to partition sets beginning with the set where the path expression begins.

First, we need a tree pattern that can match a path expression. This pattern is recursive, and the two cases are:

- Flatten (Apply (Partition (SetName() ?) ?)) - base case
- Flatten (Apply (Partition (α (? ?) ?) ?))$^{*\alpha}$ - inductive case

The predicate SetName() is true when the node represents a set which has a name. Since we want to manipulate the integer arguments to the **partition** operators, we need to name the parts of the pattern that match them:

Flatten (Apply (Partition (SetName() $a:?) ?))

Flatten (Apply (Partition (α $a:?) ?))$^{*\alpha}$

To implement the optimization, we need to find an expression where the parameters to adjacent **partition** operators are different. To do this, we need to "unroll" the pattern one level, for both the base case and the inductive case. This will give us two **partition** operators, whose arguments can be compared:

Flatten (Apply (Partition (Flatten (Apply (Partition (SetName() $a:?) ?)) $b:?) ?))

Flatten (Apply (Partition (Flatten (Apply (Partition (Flatten (? ?) $a:?)) ?) $b:?) ?))

We need one rule for each case. **Sub_select** uses the tree pattern to find a place (there will only be one) where the arguments to adjacent **partition** operators disagree. The rule then replaces the original section of plan with a section that has the correct values for the **partition** arguments. Note that all the "?"'s have been named so that the new pieces of plan can be connected to the original plan.

```
if (M := sub_select(Flatten (Apply (Partition (Flatten (Apply (Partition (SetName() $a:?)
                                          $c:?)) $b:?) $d:?)))(Q)) = {} AND $a != $b
then replace M with
         Flatten (Apply (Partition (Flatten (Apply (Partition (SetName() $a) $c)) $a) $d)))
```

```
if (M := sub_select(Flatten (Apply (Partition (Flatten (Apply (Partition (Flatten ($c:? $d:?) $a:?)
                                          $e:?)) $b:?) $f:?)))(Q)) = {} AND $a != $b
then replace M with
         Flatten (Apply (Partition (Flatten (Apply (Partition (Flatten ($c $d) $a) $e))
                                    $a) $f:?))
```

The plan that has been presented does not have any **partition**s with different parameters. For the sake of brevity, we assume that some other rule has updated the partition size of Hotels to be 3. After that rule is applied the plan would look like this:

Flatten(
    Apply(
        Partition(
            Flatten(
                Apply(
                    Partition(
                        Flatten(**Apply**(**Partition**($Hotels$, 3),
                                    $\lambda(sh)$ **Apply**($sh, \lambda(h)\, h.address$))),
                        1),
                    $\lambda(sa)$ **Apply**($sa, \lambda(a)\, a.city$))),
            1)
        $\lambda(sc)$ **Filter**($sc, \lambda(c)\, c.places\_to\_go = Theater$)))

Now the rule we have been developing would apply, and would produce the following plan, which increase the partition size of the set of addresses to 3.

$$
\begin{array}{l}
\textbf{Flatten}(\\
\quad \textbf{Apply}(\\
\quad\quad \textbf{Partition}(\\
\quad\quad\quad \textbf{Flatten}(\\
\quad\quad\quad\quad \textbf{Apply}(\\
\quad\quad\quad\quad\quad \textbf{Partition}(\\
\quad\quad\quad\quad\quad\quad \textbf{Flatten}(\textbf{Apply}(\textbf{Partition}(Hotels, 3), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda(sh)\,\textbf{Apply}(sh, \lambda(h)\,h.address))), \\
\quad\quad\quad\quad\quad\quad 3), \\
\quad\quad\quad\quad\quad \lambda(sa)\,\textbf{Apply}(sa, \lambda(a)\,a.city))), \\
\quad\quad\quad 1) \\
\quad\quad \lambda(sc)\,\textbf{Filter}(sc, \lambda(c)\,c.places\_to\_go = Theater)))
\end{array}
$$

6.2 Example 2

Our second example is the query which returns the population of any cities that have a hotel (in our database) and a theater as a tourist attraction.

$$
\begin{array}{l}
\textbf{apply}(\lambda(r)\,r.address.city.population) \\
\quad (\textbf{select}(\lambda(h)\,h.address.city.places\_to\_go = Theater)(Hotels))
\end{array}
$$

This query is interesting because it contains a pair of path expressions which denote a common subpath. It is a simple example of the situation where shared subpaths occur. Queries which are beyond the state of the art are harder to understand, and the additional complexity would obscure the presentation of our techniques. The optimization that we would like to perform is as follows: The first time that the path expression is traversed, compute a memoizing function [1] that incrementally creates a path index or cluster that associates the head-object of the shared subpath with the tail-object, allowing users of the memo structure to quickly jump from beginning to end. Subsequent traversals of the path expression will read from the memo structure and be sped up. The cost of computing and storing the memo function is small compared to the cost of jumping all around re-reading the elements of the path expression. Performing the initial translation of the query into our plan language yields:

$$\begin{aligned}&\mathbf{Apply}(\mathbf{Apply}(\mathbf{Apply}(\mathbf{Filter}(\mathbf{Apply}(\mathbf{Apply}(Hotels, \lambda(h)\, h.address),\\ &\qquad\qquad\qquad\qquad\qquad\qquad \lambda(a)\, a.city), \lambda(c)\, c.places\_to\_go = Theater),\\ &\qquad\qquad\qquad \lambda(h)\, h.address), \lambda(a)\, a.city),\\ &\quad \lambda(c)\, c.population)\end{aligned}$$

The first step in performing this optimization is locating the set of all path expressions in the query. This is accomplished using a tree query. The version of the query presented here does not account for all cases of a path expression, but is sufficient for the purposes of the example. The set of all path expressions in the query Q is:

PE(Q) = **sub_select**([Apply (α OperatorAttribute()) |
Filter (α, ContainsOperatorAttribute())]$^{*\alpha}$)(Q)

The predicate OperatorAttribute matches a node if it represents an attribute, and the predicate ContainsOperatorAttribute matches a node if the subtree rooted at the node contains an attribute. Given a set of path expressions, we want to examine each one and see if it shares a common subpath with a path P of our choosing. The **PT** operator computes the PowerTree of the tree T (see appendix A for details), and the TypeGraph function converts a tree into a tree where all non-outermost non-Apply operators are replaced by their type.

MP(P) = **select**(λ(p1) **exists**(x,p1,x n **PT**(TypeGraph(P)) and x in **PT**(TypeGraph(p1))))(PE(Q))

We want the path that has the longest shared subpath with P:

LP = **select**(λ(p1) **forall**(x,MP,length(p1) >= length(x))(MP)

We now have all the pieces that are needed to write the rule. The name assigned to the query has been used instead of writing the queries out in the action part of the rule.

```
if PE(Q) != {} then let P = choose(PE(Q)); MP; LP; P1 = choose(LP) in
                        exists($x:x,P1,x in PT(P) and x in PT(P1) and
                                  forall(y,PT(P1),length(x) >= length(y)))));
                        sub_select(Apply (α ?)*α $inp:NotApply() (? ?))(P);
                        sub_select(Apply (α ?)*α $inp:NotApply() (? ?))(P1);
                        replace sub_select($x)(P) with
                             IncrementallyBuildPathIndex(IndexName,$inp,$x)
                        replace sub_select($x)(P1) with
                             ReadPathIndex(IndexName,$inp1, $x)
```

This rule can only fire if the set of path expressions for the query is non-empty. If that is the case, it nondeterministically selects one of the path expressions to be P, and then issues the queries MP and LP. After selecting an element from LP for P1, it issues a query which allows it to name the longest matching subpath common to P and P1. It creates $inp and $inp1 as names for the root matching subpath in P and P1 respectively, and then replaces the subpaths with the appropriate code for the memoization. Note that path expressions are assumed to be rooted at a non-Apply node.

The set of path expressions that appears in example 2 is $PE(Q) =$

{ **Apply**(Hotels,address),
Apply(**Apply**(Hotels,address),city),
Filter(**Apply**(**Apply**(Hotels,address),city),places_to_go=Theater),
Apply(**Filter**(···),address),
Apply(**Apply**(**Filter**(···),address),city),
Apply(**Apply**(**Apply**(**Filter**(···),address),city),population) }

Next, we fix P to be **Filter**(**Apply**(**Apply**(Hotels,address),city),places_to_go=Theater) (From the **select**). The set of path expressions that shares a subpath with P is $MP(P) =$

{ **Apply**(**Filter**(···),address),
Apply(**Apply**(**Filter**(···),address),city),
Apply(**Apply**(**Apply**(**Filter**(···),address),city),population) }

The path with the longest shared subpath is $LP =$

{ **Apply**(**Apply**(**Apply**(**Filter**(···),address),city),population) }.

Since LP is a singleton, P is the single element of LP.

The plan for P becomes

$\mathbf{IncrementallyBuildPathIndex}(addrIndex, Hotels, Apply(Apply(Hotels, address), city))$

and the plan for P1 becomes

$$
\begin{aligned}
&\mathbf{Apply}(\mathbf{ReadPathIndex}(addrIndex,\\
&\qquad \mathbf{Filter}(\mathbf{IncrementallyBuildPathIndex}(addrIndex,\\
&\qquad\qquad Hotels,\\
&\qquad\qquad Apply(Apply(Hotels,\\
&\qquad\qquad\qquad address),\\
&\qquad\qquad city)),\\
&\qquad\quad \lambda(c)\; c.places\_to\_go = Theater),\\
&\qquad Apply(Apply(Hotels, address), city)),\\
&\quad population)
\end{aligned}
$$

Some readers might be worried by the use of PowerTree in computing this rule. It should be noted that paths longer than 6 attributes are very rare, so computing PowerTree should not be all that expensive. In addition, we have some tentative ideas about optimizing queries involving PowerTree operators.

7 Proposal for Research/Research Plan

In addition to the rule language, attribute grammar framework and optimizations presented in this document, we expect our work to continue in the following areas:

- Rule Language and Rule Control

  As discussed previously, there are a number of issues related to the control of rule application and firing, and disambiguation when multiple rules apply. We intend to provide a complete semantics for these issues. We will also examine the interaction between the plan generator's (local) control strategy and the optimizer's (global) control strategy. The rule language will be extended to take advantage of the multiple match semantics provided by using a query language for pattern matching. Writing optimization rules will give us feedback that will help us improve the rule language design.

- Hot Spot Analysis

  A particularly important rule control issue is that of determining where to focus the plan generator's attention – where to look next. We have preliminary ideas about a method for determining hot spots in queries so that the plan generator could be directed to pay particular attention to improving those areas of the plan.

- Additional Optimizations

  The optimizations presented in this document are instances of a class of optimizations. Those classes need to be explored in more detail and fleshed out. Optimizations related to clustering, more sophisticated indexing, non-set types, and joins also need to be developed.

- Analysis Information

  We expect to discover new types of useful analysis information as we develop optimizations.

- Stylized Plan Operations

  Most of the common subexpression elimination that has been discussed in this paper has been related to path expressions. There is also room at the physical level for algebraic common subexpressions [88]. Many algebra operators can be implemented by using the same algorithm with different parameters[40, 88], or as combinations of basic building block algorithms. An example illustrated in the Rexall paper[88] is the expression $\mathbf{sum}(A \cap B) - \mathbf{sum}(A - B)$. This expression can be optimized if we note that that intersection and difference can both implemented using hash join. When two operators are implemented solely in terms of a single lower level algorithm, this strategy works nicely. For operators which are implemented as combinations of lower level algorithms, this approach may not work. We would like to make the intermediate results of lower level algorithms available for use by another operator. This would allow two operators that share a common building block to avoid duplicating effort. We plan to do this by providing operators with "taps" – callback-like functional arguments which are passed to the operator. The operator invokes the taps when intermediate results are produced. This will allow us to access the hash table in the implementation of hash-join, or the intermediate sorted input to a merge join. The ability to access these intermediate results is important in the object-oriented setting, since these intermediate results can be costly to compute.

- Implementation

  This research is taking place within the EPOQ subproject of the EREQ project. Our goal for an implementation is to provide a plan generation region that could be incorporated into the EPOQ prototype to generate executable plans. These plans will then be passed to a query evaluation system based on the Persistent Object Package System (POPS) [3], developed at Brown.

  Unresolved issues regarding the implementation include the amount of the tree pattern matcher to be implemented, and the complexity of layering the appropriate primitives on top of POPS.

- Experiments

  The primary goal of performing experiments is to determine the benefit of physical level optimizations, and to characterize the kinds of queries that benefit the most. Also, experimental data will help direct our search for new physical level optimizations. There are two kinds of experiments that could be performed. The first will tell us whether or not there is any payoff (and how much if there is) for doing optimizations at the physical level. For these experiments we will optimize a set of queries twice, the first time with physical level optimizations turned on, the second time with them turned off. The second type of experiment is to compare the performance of our system with another plan generation system, comparing the quality of the generated plan, and the amount of time required for optimization. Unfortunately, it is very hard to do a fair or meaningful comparison of systems, because few systems use a common query language, plan language or query execution engine. We are also concerned with the efficiency of optimization, and the experiments should allow us to see the impact of the sophisticated rule language that we have proposed.

8 Conclusions

We expect the contributions of thesis to include a broaden of our understanding of Physical Level Optimization – what can and cannot be accomplished. The introduction of a rule language that uses a query language for pattern matching is another contribution, along with the implied notion that the optimizer can optimize its own internal queries. The application of attribute grammars for computing analysis information and the specific analysis information will add to our knowledge of ways to optimize queries. Use of the **partition** operator to control path expression traversal order and buffer pool usage is also expected to improve the performance of queries. The development of hot-spot analysis and stylized plan operators seems promising but at this point these ideas are still tentative. Our catalog of physical optimizations will expand the repertoire of techniques available to database system implementors.

Even at this juncture, we have some ideas concerning future work beyond the scope of the thesis. Cost Models for object-oriented systems continue to be an important area of research, and would improve the accuracy of our physical optimization techniques. The ability to optimize physical level structures suggests that there may be opportunities to optimize queries that manipulate more than one bulk types, since many distinctions that are made at the source level disappear at the physical level. Related to this idea is the notion of freely intermixing source level and physical level optimizations, and how to control such interleaving. Research on control strategies for optimizers is becoming more interesting – it may be interesting to have sets of control strategies which apply only to certain kinds of queries. A particularly interesting question is whether more of the semantic information available at the source level can be brought down the the physical level, so that optimizations that were formerly thought of as source level optimizations could be performed at the physical level. As always in the field of optimization, the search for new optimization rules is an ongoing matter of research.

Acknowledgments: The work that is described in this document was performed jointly with Professor Stanley B. Zdonik at Brown University. The work on the AQUA query language was performed in a group with Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The tree query language was developed in a group with Bharathi Subramanian, Scott L. Vandenberg, and Stanley B. Zdonik.

A Operators for Tree Query Algebra

- **select**(p)(T) - return a list of subgraphs of T by finding all the nodes in T that satisfy P and maintaining the transitive parent-child relationship.
- **sub_select**(r)(T) - return the set of subgraphs of T which match pattern r.
- **PT**(T) - return the set of all subtrees (PowerTree) of T.
- **PSG**(T) - return the power subgraph of T — the set of all connected subgraphs of T.

- **all_anc(r)**(T) - returns the set of maximal subtrees of T that end with the pattern r.
- **all_desc(r)**(T) - returns the set of maximal subtrees of T that begin with the pattern r.
- **im_ancestor**(x)(T) - find the immediate ancestor of x in T.
- **im_descendant**(x)(T) - find the set of immediate descendants of x in T.
- **tree**(x) - create a tree from x.
- **root**(T) - return the root node of T.
- **leaves**(T) - return the set of leaves of T.
- **nodes**(T) - return the set of nodes in T.
- **append**(parent,sibling)(T1,T2) - Append T1 to T2 as a child of the parent node, after the subtree rooted at sibling.
- **add_node**(x,parent,sibling)(T) - add x to T, as a child of the parent node, after the subtree rooted at sibling.
- **replace_node**(x,y)(T) - replace node x in T with node y

B Attribute Grammar for Dependency Information

We demonstrate our notation by presenting an attribute grammar that computes simple dependency information. First, we present a version for the logical algebra, followed by a version for the physical algebra.

Q {Q.IndirectlyDependent = Q0.DirectlyDependent;
  Q.DirectlyDependent=Q0.DirectlyDependent} ::= apply(F)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= select(P)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= exists(P)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= forall(P)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= mem(O,EQ)(Q) {}.

Q {Q.IndirectlyDependent = Q0.DirectlyDependent;
Q.DirectlyDependent = Q0.DirectlyDependent} ::= fold(u,F,c)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= union(EQ,T)(Q,Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= intersect(EQ,T)(Q,Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent;
Q.IndirectlyDependent = Q1.DirectlyDependent ∪ Q0.IndirectlyDependent}
::= diff(EQ,T)(Q,Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= LFP(EQ,F)(Q) {}.

Q {} ::= set(O) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= choose(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= group(F)(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= dup_elim(EQ)(Q) {}.

Q {} ::= nest(L)(Q) {}.

Q {} ::= unnest(L)(Q) {}.

Q {} ::= convert(Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;

Q.IndirectlyDependent = {}} ::= join(P,F)(Q,Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= tup_join(P,F)(Q,Q) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= outer_join(P,F,F,F,T)(Q,Q) {}.

Q {Q.DirectlyDependent = <collection-name>; Q.IndirectlyDependent = {}}
::= <collection-name> {}.

T {} ::= TypeExpr {}.

F {} ::= lambda <var> Q {}.

F {} ::= lambda <var> Path {}.

F {} ::= lambda <var> Expr {}.

P {} ::= <function> {}.

L {} ::= <set of tuple labels> {}.

O {} ::= <object-designator> {}.

Path {} ::= NamePath {}.

NamePath {} ::= <attributeName> {}.

NamePath {} ::= NamePath <attributeName> {}.

The dependency attribute grammar for the physical algebra is:

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= Partition(Q,int) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= Filter(Q,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent;
Q.IndirectlyDependent = Q0.DirectlyDependent} ::= Apply(Q,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}

::= IndexFilter(Q,N,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= HashFilter(Q,N,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= Join(Q,Q,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= LoopJoin(Q,Q,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= MergeJoin(Q,Q,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent ∪ Q1.DirectlyDependent;
Q.IndirectlyDependent = {}} ::= HashJoin(Q,Q,H,F) {}.

Q {Q.DirectlyDependent = Q0.DirectlyDependent; Q.IndirectlyDependent = {}}
::= Flatten(Q) {}.

Q {Q.DirectlyDependent = N.DirectlyDependent;
Q.IndirectlyDependent = Q1.DirectlyDependent}
::= IncrementallyBuildPathIndex(N,Q,TP) {}.

Q {Q.DirectlyDependent = N.DirectlyDependent;
Q.IndirectlyDependent = Q1.DirectlyDependent}
::= IncrementallyBuildCluster(N,Q,TP) {}.

Q {Q.DirectlyDependent = N.DirectlyDependent;
Q.IndirectlyDependent = Q1.DirectlyDependent}
::= ReadPathIndex(N,Q) {}.

Q {Q.DirectlyDependent = N.DirectlyDependent};
Q.IndirectlyDependent = Q1.DirectlyDependent}
::= ReadCluster(N,Q) {}.

Q {Q.DirectlyDependent = N.DirectlyDependent; Q.IndirectlyDependent = {}}
::= N {}.

F {F.DirectlyDependent = Q.DirectlyDependent; Q.IndirectlyDependent = {}}
::= lambda AL Q {}.

F {F.DirectlyDependent = P.DirectlyDependent; Q.IndirectlyDependent = {}}
::= lambda AL PL P {}.

F {F.DirectlyDependent = E.DirectlyDependent; Q.IndirectlyDependent = {}}

::= lambda AL E {}.

E {E.DirectlyDependent = E1.DirectlyDependent ∪ P.DirectlyDependent;
E.IndirectlyDependent = {}} ::= E op P {}.

E {E.DirectlyDependent = N.DirectlyDependent; E.IndirectlyDependent = {}} ::= N {}.

E {E.DirectlyDependent = P.DirectlyDependent; E.IndirectlyDependent = {}} ::= P {}.

AL {AL.DirectlyDependent = {TypeOf(var)}; Q.IndirectlyDependent = {}} ::= var {}.

AL {AL.DirectlyDependent = {TypeOf(var)} ∪ {TypeOf(var)}; Q.IndirectlyDependent = {}}
::= AL , var {}.

PL {PL.DirectlyDependent = AL.DirectlyDependent; Q.IndirectlyDependent = {}} ::= (AL) {}.

N {Q.DirectlyDependent = {identifier}; Q.IndirectlyDependent = {}} ::= identifier {}.

TP {TP.DirectlyDependent = {TypeOf(identifier)}; Q.IndirectlyDependent = {}} ::= identifier {}.

TP {TP.DirectlyDependent = TP1.DirectlyDependent ∪ {TypeOf(var)};
Q.IndirectlyDependent = {}} ::= TP . identifier {}.

P {P.DirectlyDependent = {TypeOf(var)};
Q.IndirectlyDependent = {}} ::= var . TP {}.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[2] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[3] Lalit Agrawal, Sean Cutts, Jennifer Goree, Marc Hamilton, Eric Lam, Neal Pawar, Adam Stauffer, and Rodrigo Vanegas. ?????POPS design document. CS191 Final Project Report, Brown University, May 1993.

[4] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[6] Peter M.G. Apers and Gio Wiederhold, editors. *Proceedings of the Fifteenth Very Large Databases Conference*, Amsterdam, The Netherlands, 1989. Morgan Kaufmann Publishers, Inc.

[7] Catriel Beeri, editor. *Proceedings of the The Fourth International Workshop on Database Programming Languages*, New York, New York, August 1993. Springer-Verlag.

[8] Veronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in O2. In Dearle et al. [27], pages 403–412.

[9] Elisa Bertino. An indexing technique for object-oriented databases. In *Proceedings of the Seventh International Conference on Data Engineering* [48], pages 160–170.

[10] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[11] Jia bing R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. In Clifford and King [18], pages 22–31.

[12] Jose A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences building the open OODB query optimizer. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 287–296, Washington D.C., May 1993. ACM Special Interest Group on Management of Data, ACM Press.

[13] Chee Yong Chan, Beng Chin Ooi, and Hongjun Lu. Extensible buffer management of indexes. In Yuan [92], pages 444–454.

[14] Ellis E. Chang and Randy H. Katz. Expliting inheritance and structure semantics for effective clustering and buffering in an object-oriented dbms. In Clifford et al. [19], pages 348–357.

[15] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[16] Chi-Hung Chi and Hank Dietz. Unified management of registers and cache using liveness and cache bypass. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 344–355, Portland, Oregon, June 1989. ACM Special Interest Group on Programming Languages, ACM Press. Also appears as SIGPLAN Notices Vol. 24 No. 7, July 1989.

[17] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of the Eleventh Very Large Databases Conference*, pages 127–141, Stockholm, August 1985. VLDB, Morgan Kaufmann Publishers, Inc.

[18] James Clifford and Roger King, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991. ACM Special Interest Group on Management of Data, ACM Press.

[19] James Clifford, Bruce Lindsay, and David Maier, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989. ACM Special Interest Group on Management of Data, ACM Press.

[20] Sophie Cluet and Claude Delobel. Towards a unification of rewrite based optimization techniques for object-oriented queries. Got from GMS, February 1991.

[21] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Beeri [7], pages –.

[22] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[23] Thomas A. Cooper and Nancy Wogrin. *Rule-based Programming with OPS5*. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988.

[24] Douglas W. Cornell and Philip S. Yu. Integration of buffer management and query optimization in relational database environment. In Apers and Wiederhold [6], pages 247–255.

[25] Scott Daniels, Goetz Graefe, Thomas Keller, David Maier, Duri Schmidt, and Bennet Vance. Query optimization in Revelation, an overview. *Quarterly Bulletin of the IEEE Computer Society technical committee on Data Engineering*, 14(2):58–62, June 1991.

[26] Umeshwar Dayal and Irv Traiger, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, San Francisco, California, May 1987. ACM Special Interest Group on Management of Data, ACM Press. Also Appears as SIGMOD Record, Vol. 16 No. 3, December 1987.

[27] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Proceedings of the Fourth International Workshop on Persistent Object Systems – Implementing Persistent Object Bases:Principles and Practice*, Martha's Vineyard, MA, September 1990.

[28] C. Delobel, M. Kifer, and Y. Matsunaga, editors. *Proceedings of the The Second International Conference on Deductive and Object-Oriented Databases*, number 566 in Lecture Notes in Computer Science, Munich, Germany, December 1991. Springer-Verlag.

[29] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[30] Christos Faloutso, Raymond Ng, and Timos Sellis. Predictive load control for flexible buffer allocation. In Lohman et al. [65], pages 265–274.

[31] Leonidas Fegaras, David Maier, and Tim Sheard. Specifying rule-based query optimizers in a reflective framework. In *Proceedings of the The Third International Conference on Deductive and Object-Oriented Databases*, Scottsdale, Arizona, December 1993.

[32] Charles N. Fischer and Jr Richard J. LeBlanc. *Crafting a Compiler in C*. Benjamin Cummings Publishing Company, Inc, Menlo Park, California, 1991.

[33] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[34] Johann Christoph Freytag. A rule-based view of query optimization. In Dayal and Traiger [26], pages 173–180. Also Appears as SIGMOD Record, Vol. 16 No. 3, December 1987.

[35] Johann Christoph Freytag and Nathan Goodman. Translating aggregate queries into iterative programs. In Wesley Chu, Georges Gadarin, Setsuo Ohsuga, and Yahko Kambayashi, editors, *Proceedings of the Twelfth Very Large Databases Conference*, pages 138–146, Kyoto, Japan, August 1986. VLDB, Morgan Kaufmann Publishers, Inc.

[36] Johann Christoph Freytag and Nathan Goodman. On the translation of relational queries into iterative programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.

[37] Hector Garcia-Molina and H.V. Jagadish, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990. ACM Special Interest Group on Management of Data, ACM Press.

[38] Georges Gardarin and Rosana S. G. Lanzelotte. Optimizing object-oriented database queries using cost-controlled rewriting. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the Third International Conference on Extending Database Technology*, number 580 in Lecture Notes in Computer Science, pages 534–549, Vienna, Austria, March 1992. Springer-Verlag.

[39] Goetz Graefe. *Rule-Based Query Optimization in Extensbile Database Systems*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, November 1987.

[40] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, pages 73–170, June 1993.

[41] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In Dayal and Traiger [26], pages 160–172. Also Appears as SIGMOD Record, Vol. 16 No. 3, December 1987.

[42] Goetz Graefe and Willam J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ningth International Conference on Data Engineering*, pages 209–218, Vienna, Austria, April 1993. IEEE, IEEE Computer Society Press.

[43] Goetz Graefe and William McKenna. The Volcano optimizer generator. Technical Report 563, University of Colorado at Boulder, December 1991.

[44] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In Clifford et al. [19], pages 358–366.

[45] Goetz Grafe, David Maier, Scott Daniels, and Tom Keller. A software architecture for efficient query processing in object-oriented database systems with encapsulated behavior. Got via FTP.

[46] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[47] Laura M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In Clifford et al. [19], pages 377–388.

[48] IEEE. *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991. IEEE Computer Society Press.

[49] Anant Jhingran and Michael Stonebraker. Alternatives in complex object representation: A performance perspective. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 94–102, Los Angeles, California, February 1990. IEEE, IEEE Computer Society Press.

[50] Paris Kanellakis and Joachim W. Schmidt, editors. *Bulk Types & Persistent Data: The Third International Workshop on Database Programming Languages*, Nafplion, Greece, August 1991. Morgan Kaufmann Publishers, Inc.

[51] Tim Keller, Goetz Graefe, and David Maier. Efficient assembly of complex objects. In Clifford and King [18], pages 148–157.

[52] Tom Keller and Goetz Graefe. The one-to-one match operator of the Volcano query processing system. Technical Report CS/E 89-009, Oregon Graduate Institute of Science and Technology, June 1989.

[53] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function materialization in object bases. In Clifford and King [18], pages 258–267.

[54] Alfons Kemper and Guido Moerkotte. Access support in object bases. In Garcia-Molina and Jagadish [37], pages 364–376.

[55] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 290–301, Brisbane, Australia, August 1990. Morgan Kaufmann Publishers, Inc.

[56] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. A blackboard architecture for query optimization in object bases. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 543–554, Dublin, Ireland, August 1993. Morgan Kaufmann Publishers, Inc.

[57] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[58] R. S. G. Lanzelotte, P. Valduriez, M. Ziane, and J.-P. Cheiney. Optimization of nonrecursive queries in OODBs. In Delobel et al. [28], pages 1–21.

[59] Rosana S. G. Lanzelotte and Jean-Pierre Cheiney. Adapting relational optimization technology to object-oriented and deductive database languages. In Kanellakis and Schmidt [50], pages 322–336.

[60] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zait. Optimization of object-oriented recursive queries using cost-controlled strategies. In Stonebraker [79], pages 256–257.

[61] Mavis K. Lee, Johann Cristoph Freytag, and Guy M. Lohman. Implementing an interpreter for functional rules in a query optimizer. In Francois Bancilhon and David J. DeWitt, editors, *Proceedings of the Fourteenth Very Large Databases Conference*, pages 218–229, Los Angeles, California, August 1988. VLDB Endowment, Morgan Kaufmann Publishers, Inc.

[62] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The AQUA data model and algebra. In Beeri [7].

[63] Daniel F. Lieuwen and David J. DeWitt. Optimizing loops in database programming languages. In Kanellakis and Schmidt [50], pages 287–305.

[64] Guy H. Lohman. Grammar-like functional rules for representing query optimization alternatives. In Haran Boral and Per ake Larson, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 18–27, Chicago, Illinois, June 1988. ACM Special Interest Group on Management of Data, ACM Press. Also appears as SIGMOD Record Vol. 17 No. 3, September 1988.

[65] Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors. *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), August 1991. Morgan Kaufmann Publishers, Inc.

[66] Andrew C.T. MacKeith. Ereq query representation and cost model. Master's thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910, May 1993.

[67] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In Carlo Zaniolo, editor, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 84–95, Washington, D.C., June 1986. ACM Special Interest Group on Management of Data, ACM Press. Also Appears as SIGMOD Record, Vol. 15 No. 2, June 1986.

[68] David Maier and Jacob Stein. Indexing in an object-oriented database. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 171–182, Pacific Grove, California, September 1986. Association for Computing Machinery.

[69] Gail Mitchell. *Extensible Query Processing in an Object-Oriented Database.* PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910, May 1993.

[70] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. An architecture for query processing in persistent object stores. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences.* IEEE, IEEE Computer Society Press, January 1991.

[71] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In Clifford and King [18], pages 387–396.

[72] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. In Stonebraker [79], pages 403–412.

[73] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In Stonebraker [79], pages 39–48.

[74] Giovanni Maria Sacco and Mario Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the Eighth International Conference on Very Large Databases*, pages 257–262, Mexico City, Mexico, September 1982. VLDB, Morgan Kaufmann Publishers, Inc.

[75] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, December 1986.

[76] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 23–34. ACM Special Interest Group on Management of Data, ACM Press, 1979.

[77] Karen Shannon and Richard Snodgrass. Semantic clustering. In Dearle et al. [27], pages 389–402.

[78] Eugene J. Shekita and Michael J. Carey. A performance evaluation of pointer-based joins. In Garcia-Molina and Jagadish [37], pages 300–311.

[79] Michael Stonebraker, editor. *Proceedings of the SIGMOD International Conference on Management of Data*, San Diego, California, June 1992. ACM Special Interest Group on Management of Data, ACM Press.

[80] Michael Stonebraker, Eugene Wong, Peter Greps, and Gerald Held. The design and implementation of Ingres. *ACM Transactions on Database Systems*, 1(3):180–222, September 1976.

[81] Dave D. Straube and M. Tamer Ozsu. Execution plan generation for an object-oriented data model. In Delobel et al. [28], pages 43–67.

[82] David D. Straube. *Queries and Query Processing in Object-Oriented Database Systems*. PhD thesis, Univeristy of Alberta, December 1990.

[83] David D. Straube and M. Tamer Ozsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[84] Bharathi Subramanian, Stanley B. Zdonik, Theodore W. Leung, and Scott L. Vandenberg. Ordered types in the AQUA data model. In Beeri [7].

[85] Manolis M. Tsangaris and Jeffrey F. Naughton. A stochastic approach for clustering in object bases. In Clifford and King [18], pages 12–21.

[86] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In Stonebraker [79], pages 144–153.

[87] Bennet Vance. Towards an object-oriented query algebra. Technical Report CS/E-91-008, Oregon Graduate Institute of Science and Technology, January 1992.

[88] Bennet Vance. An abstract object-oriented query execution language. In Beeri [7], pages –.

[89] Scott L. Vandenberg. *Algebras for Object-Oriented Query Languages*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, June 1993.

[90] Mary Jane Willshire. How spacey can they get? space overhead for storage and indexing with object-oriented databases. In *Proceedings of the Seventh International Conference on Data Engineering* [48], pages 14–22.

[91] Eugene Wong and Karel Youssefi. Decomposition – a strategy for query processing. *ACM Transactions on Database Systems*, 1(2):223–241, September 1976.

[92] Li-Yan Yuan, editor. *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992. Morgan Kaufmann Publishers, Inc.

[93] Stanley B. Zdonik. Query optimization in object oriented databases. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 19–25. IEEE, IEEE Computer Society Press, January 1989.