Semantic Query Optimisation and Rule Graphs

Jerome Robinson
Computer Science Department
University of Essex, UK.
robij@essex.ac.uk

Barry Lowden
Computer Science Department
University of Essex, UK.
lowdb@essex.ac.uk

Abstract

Previous researchers note the problem for semantic optimisation of database queries caused by its production of a large number of semantically equivalent alternative queries, from which one must be quickly chosen. The present paper observes that this is caused by the non-uniform structure of rules, which also (i) prevents fast access to rules, (ii) allows the rule base to become impractically large, and (iii) such rules anyway are of low utility for SQO. Simple rules are less expressive but more useful in this application. Single-antecedent, single-consequent rules are the main component of the rule base. The rule set then constitutes a graph whose edges are the rules. This Condition Dependency graph provides a map of possible query reformulation operations so that guided direct modification of the single existing query is possible instead of blind sequential application of rules and cost evaluation for the resulting set of alternative queries.

1 Introduction

A semantic query optimiser is an intelligent interface between user and database, which intercepts the user's query before it reaches the data server. The optimiser uses its knowledge of the data to improve the query and then forwards a reformulated version of the query to the database. Query Q' is a *semantic reformulation* of query Q if Q' is different from Q, and Q' cannot be derived from Q by syntactic operations, and Q' gives the same query result as Q. Query Q' must be faster for the server to process, then Q. Query processing speed is measured by *cost assessment* using cost factors such as tuple retrieval time from disk, and comparison time per tuple to compute query conditions. The *cost improvement* produced by modification of an existing query is much easier to assess than the *absolute* cost of a new query. In fact, many *types* of modification will improve query cost and there is no need to spend time computing the *amount* of each specific improvement.

Rules describing the data are used to transform the query. Integrity Constraint rules were used in early systems with limited success, but more recently rule discovery from the data itself has been used to provide more *query-relevant* knowledge of the data. Automatic knowledge discovery can lead to large rule sets (much larger than the data sets they describe), whereas a set much smaller than the data is required. Previous workers [*e.g.* Hs 95, Lo 95, Si 88] have, in effect, used *a sample* from the potentially derivable rule set, by generating a few rules in response to each query. But this can provide a *random* sample with low probability of containing useful rules. The histogram-based rule sets described in this paper are small, easily accessed, and all rules have a good chance of being used.

Previous semantic optimisation algorithms [*e.g.* Hs 95, Ki 81, Sh 88, Si 88] have been *iterative*, progressively applying new rules as the query is changed by previous rules. ('Rule application' means adding consequent conditions to the query from rules whose antecedents are implied by query conditions). This is slow. Its *sequential* character is undesirable in a process which must be as fast as possible, to avoid delaying the query and counteracting time saved when the data server processes the improved query. The Dilution Effect identified in section 4, suggests that successive rule applications provide information of decreasing quality, which is progressively less likely to be useful in query reformulation.

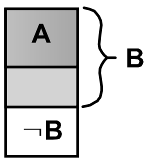
The rulebase partitions easily into subsets of current query interest, and the rule graph for each subset can be pre-processed, allowing time-consuming decisions to be made before queries arrive, so that fast lookup replaces the iterative process and its subsequent time-consuming cost evaluation.

The rules discussed in this paper are basically if/then expressions which contain two predicates on attributes of a relational database table. For example: $(15 \le a \le 30) \Rightarrow (243 \le d \le 271)$, which means "if the value of attribute 'a' in a tuple is in the range 15 to 30 then the value of attribute 'd' will be in the range 243 to 271". In SQO the antecedent condition(s) of rules are matched with query conditions. Since *all* antecedent conditions in a rule must be matched, rules with only a *single* antecedent are most

Proceedings of the 5th KRDB Workshop
Seattle, WA, 31-May-1998
(A. Borgida, V. Chaudhri, M. Staudt, eds.)
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/

widely usable. The two predicates (antecedent and consequent) in each rule are Selection Conditions or constraints of the type found in database queries. Each condition therefore denotes a subset of a database table. Rules are cascaded using subrange containment as the link between successive rules.

A rule $A \Rightarrow B$ partitions the set of tuples in a database relation into three parts:



This has a number of implications. Eg: (i) the rule graph is a Hasse Diagram for the Ordered Set of tuple subsets denoted by potential query conditions; (ii) the nodes in a graph path denote monotonically increasing subsets of the data set; (iii) the Dilution Effect; (iv) the potential usefulness of a rule for SQO can be quantified using the relative sizes of set(A) and set(B).

The structure of the paper is as follows. Section 2 identifies the main operations used in semantic query transformation. Section 3 introduces the Graph interpretation of semantic optimisation. Section 4 specifies fundamental properties of rules and paths in the graph (a path with two or more edges corresponds to two or more cascaded rules). Section 5 indicates the potential size of an unconstrained rulebase, and Section 6 introduces histogram-based multi-consequent rules as an efficient implementation for attribute-pair rule sets, which reduces the number of rule sets and look-ups, and can be derived by a single scan through a database relation.

2 Background

The purpose of query optimisation is faster response to queries. The semantic optimiser knows more about its data than the user. It can therefore replace the user's query with a different query which will produce exactly the same result set, but faster. The new query is faster because it will do less work when extracting the selected result tuples from the database

2.1 Semantic Query Processing Operations

This section indicates methods traditionally used in SQO. A query is canonically a sum of products of selection conditions. It is semantically optimised by processing the conjunctive sub-queries, using the following operations.

2.1.1. Condition Elimination

If a query contains conditions G and H, and a rule $G \Rightarrow H$ exists (*meaning:* if G is true in any tuple then H is also true) then there is no need for the data server to test both conditions during query processing. The implied condition can be eliminated from the query.

2.1.2. Query Refutation by Condition Contradiction

If a query contains the conditions (g = "propane") and ($100 \leq h \leq 200$), but rule (g = "propane") $\Rightarrow$ ($13 \leq h \leq 71$) exists, then it is impossible for both query conditions to be satisfied by any tuple. The query will therefore produce no results, and can be answered immediately with the empty set, without consulting any data.

2.1.3. Query Refutation by Contradictory Consequents

If two query conditions match the antecedents of two different rules whose consequent conditions are contradictory, then the query will produce no results and can be answered with the empty set. The two query conditions select disjoint sets of tuples from the database table. (A conjunctive query identifies the *intersection* of condition-selected sets).

2.1.4. Query Refutation by Empty Set Selection

If any query condition selects an empty set then the conjunctive query will produce the empty set as result. A query condition can be identified as an empty set condition if it specifies a range of values for an attribute which is known to be outside the two extreme values for that attribute, or specifies values in a known Empty Subrange Interval between the two extreme values, or it specifies an equality condition using a string value known to be absent from a category attribute.

2.1.5. Condition Elimination by Full Set Selection

Any query condition which selects all the tuples in the database table is redundant. A range condition whose range interval includes both extreme values for the attribute is an example of a full set condition, which can be eliminated from the query.

2.1.6. Condition Range Narrowing

When one query condition implies a subrange of the range specified in another query condition (by range overlap or nesting) then narrowing the implied query condition range allows faster elimination of intermediate tuples during query processing in the data server. Later (and therefore probably more expensive) query conditions in the data server's query execution plan will then have fewer tuples to test.

2.1.7. Condition Addition for Index Introduction

An index provides direct access to relevant disk blocks. If this reduces the number of blocks accessed to answer the query (because the query relevant tuples are concentrated in a relatively small number of blocks) then query processing time is shortened. Some data servers are able to make use of *multiple* indexed attributes to retrieve a single set from a single database relation, in which case multiple indexed conditions can usefully be added to the query.

2.1.8. Condition Addition for Subset Selection

If a query consists of a single high-cost condition it is desirable to reduce the number of tuples to which this condition must be applied. Adding an indexed condition may help, but in some circumstances a low-cost nonindexed condition implied by the high cost condition can be usefully added to the query, to act as a tuple filter (selector) for the high-cost condition.

2.1.9. Empty Projection

Eg: A query requests the cost of houses in region NW3.
A rule in the *region* $\longrightarrow$ *cost* attribute-pair rule set states: (region = NW3) $\Rightarrow$ no tuples.

This means that houses do exist in the database with values (region = NW3) but all have null values in the "cost" field. Therefore the query will have no result values and can be answered directly from the rule, without consulting the data.

Previous knowledge bases did not support this operation because the absence of tuples caused an absence of rules. But the knowledge representation proposed in section 6.1 is *an extended histogram* for the antecedent attribute. Each bar in the histogram identifies a sub-set of the data, and multiple consequents describe that sub-set. Because of *null values* in particular fields in the data tuples, some attributes in the consequent vector may report "no tuples" while others specify attribute values.

2.2 Knowledge Representation Requirements

The domain of a database is large. Therefore *descriptive* knowledge must satisfy the following conditions:

(i) The rule set must be *modular* so that rules can be used, derived and discarded in conveniently manageable blocks (subsets).

(ii) Rule set modules must be derivable in real time. A new user's query indicates the data in which they are interested. Relevant rules sets must be derivable before the user's next query arrives. Since rules are easily produced they are also easily *discarded* without lengthy consideration. (The modularity allows rule set caching, so that rules which show decreasing use by current queries are retained in secondary memory until query interest has definitely moved to other areas of the data. The rule subset is then discarded in order to prevent unnecessary accumulation, and to avoid the need to *update* passive rules when the data they describe is changed).

2.3 The Knowledge Base

For query reformulation there are three main types of knowledge derived from the data, namely domain assertions, attribute-pair rules and special-purpose rules.

Domain assertions are single-attribute assertions such as the two extreme values, empty subrange intervals between the extreme values, percentile points, string attribute values and frequencies.

Attribute-pair rules (called Simple Rules in [Si 88]) are single antecedent rules which also have a single consequent condition. Each attribute-pair (AP) rule refers to two columns in a particular database table, which may be a Joined table from two or more base relations. Each rule is an inference of the form $A \Rightarrow B$ where A and B are single conditions (atomic constraints) of the kind found in queries, and each refers to a different attribute in the same database table. Conditions include *equality conditions* such as (category = 'paper') or *range conditions* on numeric types, such as (date $\leq$ 9.1.98).

Each of the two conditions in an AP rule is labelled with the percentage of the database table it denotes. This is the percentage of tuples the condition would extract if it were used as a selection condition in a query.

Special-purpose rules are derived to service specific frequent queries or query condition combinations. A rule such as: (a = "Seattle") $\wedge$ ($50 \leq c \leq 100$) $\Rightarrow$ ($37 \leq g \leq 81$) is potentially useful if the specified antecedent conditions *often occur together* in queries. This is analogous to a multi-attribute key, which selects a single tuple by specifying an equality condition on more than one attribute. Similarly, data subsets of regular interest can each be identified by a specific conjunction of conditions. Since a *fixed* pair of conditions is used, the pair can be regarded as a single unit, and does not lead to a combinatorial explosion in the size of the rule set (by cartesian product combination of values from the two attributes). Only specific multi-attribute conditions which are used often are of interest as rule antecedents. *All* antecedent conditions in a rule must be matched with query conditions in order to use the rule for query optimisation, so multi-condition-antecedent rules are not useful for *general* query processing. The probability that a query contains the specific combination of conditions in a rule antecedent decreases as the number of antecedents in the rule increases. So a large number of rules would need to be held in order to improve the hit ratio for queries. Single-antecedent rules are more likely to be used and have much smaller maximum set size. So single-antecedent rules should be the main component of the rulebase in SQO systems.

3 Rule Graphs

An Attribute-pair rule is a pair of conditions linked by a directed arc. Such a rule can be used as an edge in a directed graph, whose connected paths denote chains of inference. Two rules can be cascaded to form a path if the consequent of one implies the antecedent of the other. For *equality* conditions this means the two conditions match exactly, whereas for two *range* conditions RC_1 and RC_2 on the same attribute, RC_1 implies RC_2 if the range in RC_1 is a subrange of that in RC_2. *Subrange containment* is therefore the rule of inference that links consecutive rules. Subranges select subsets, so the set of tuples selected by the consequent of one rule is a subset of the set selected by the other rule's antecedent. Therefore the consequent *assertion* satisfies the antecedent *constraint* in the second rule.

Query conditions identify particular condition nodes in the graph, by matching rule antecedent or consequent conditions. Certain paths attached to these *distinguished nodes* are of interest for the SQO operations listed in section 2.1 above:

Operation 2.1.1 needs a path between two distinguished nodes (such as the path A to E in Figure 1). Query conditions are usually matched with rule *antecedent* conditions, but in operation 2.1.1, one of the two query conditions linked by a graph path is *implied by* a rule *consequent*. For all the other operations a query condition is the *start* of a path rather than the end, since we are interested in what the query condition *implies*.

Operation 2.1.2, for example, needs paths to any conditions on attribute K, where K is restricted in the query. Operation 2.1.3 compares assertions reachable from different query nodes. Operation 2.1.7 needs a path to an indexed attribute condition, while operation 2.1.8 needs a path from a high cost query condition one whose attribute *type* has lower comparison cost per tuple.

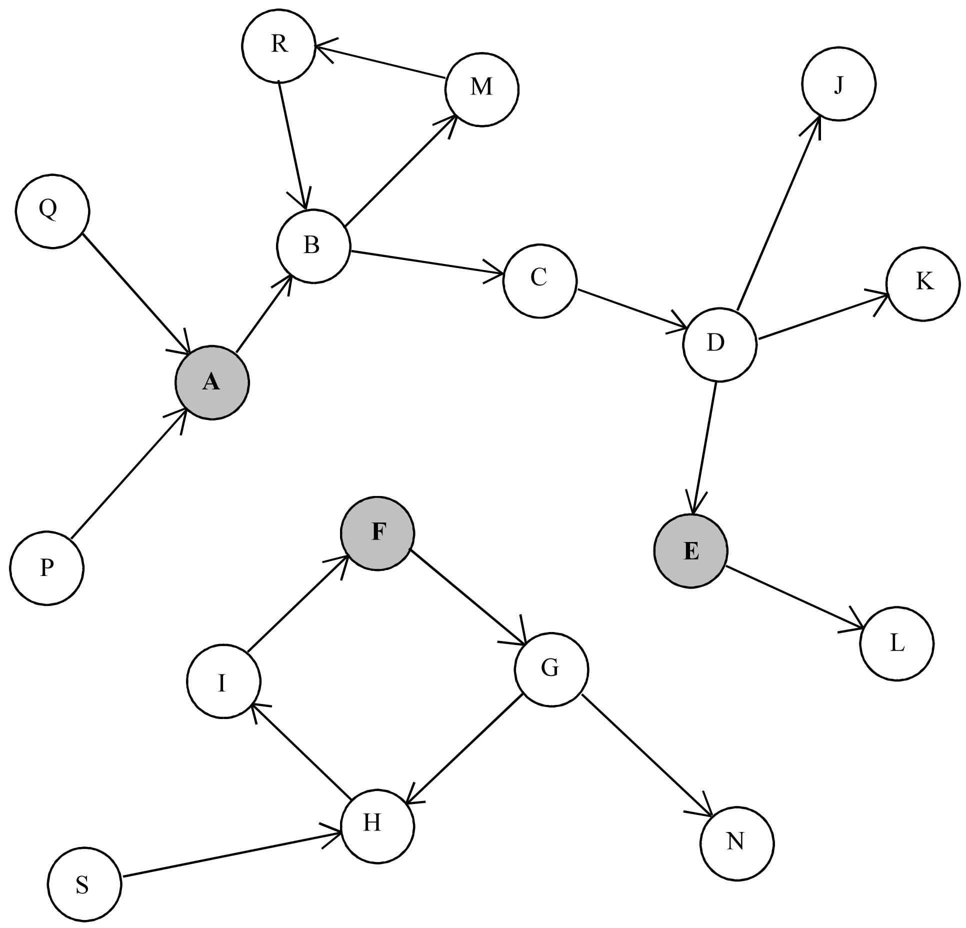
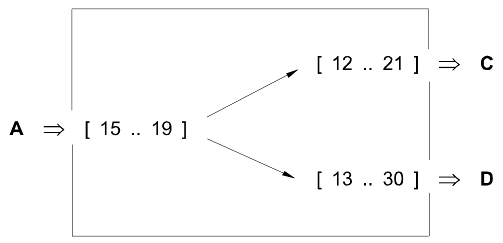


Figure 1. Part of a Rule Graph showing query conditions A, F and E.

A cycle (such as B, M, R in Figure 1) denotes a set of equivalent conditions on different attributes. All conditions in a cycle select the same set of tuples. Conditions in a query can be replaced by equivalent conditions which are faster to process. E.g. a string attribute which differs in more than the first eight characters requires more than one CPU step to compare with a specified value. Replacing the string attribute condition with a different condition on a simple numeric attribute will reduce the time needed per tuple to process the query.

When rules are cascaded by subrange containment rather than exact match, their *intermediate graph node denotes a set of conditions*, as shown inside the box in the diagram on the right. The consequent range [15 .. 19] matches (i.e. implies) any rule antecedent of which it is a sub-range. The box denotes a graph node. All ranges inside the box refer to the same attribute.

Branches occur in graph paths when a consequent range is a subrange of more than one antecedent range. The multiple antecedents included in one of these composite graph nodes must have *overlapping ranges* since the overlap region *contains* the consequent node range (e.g. [15 .. 19] above).



Branches also occur when the same antecedent condition appears in more than one rule, whose consequents describe different attributes. Eg:

(a = "PC-49") $\Rightarrow$ ($115.04 \le b \le 208.63$)

and (a = "PC-49") $\Rightarrow$ ($3.6.80 \le c \le 12.7.82$).

The traditional approach to semantic query reformulation [Eg. Sh 88] does not refer to a graph. It entails iteratively adding consequent conditions from rules to the query if the rule antecedents are implied either by Original Conditions in the query or by conditions subsequently added to the query. The process, known as Semantic Expansion, can be seen as path building in the Rule Graph. Path building is necessarily a *sequential* process, which is therefore undesirably slow for SQO. A single-step process is preferable. Furthermore, each added condition from each rule application produces a *new query*, semantically equivalent to the original. After a certain amount of expansion, the traditional approach examines the set of queries it has produced, computes the execution cost of each, and forwards the lowest cost query to the data server to process (*i.e.* answer). This cost-computation phase is also an unsuitably slow activity for a process in a hurry.

Figure 1 clarifies the process of semantic query optimisation, revealing it as modification of an existing query, using graph paths, rather than the sequential generation of multiple alternative queries. Of the nine query processing operations listed in section 2, only operations 7 and 8 require any cost assessment. The others involve only path connectivity or domain assertion checking. A knowledge of path and graph characteristics, as discussed below, is therefore relevant.

Theorem 1: A Path denotes a Sequence of Nested Sets, Monotonically Increasing in Size.

Proof: Each condition in a rule denotes a set of tuples. This is the set obtained if the condition is used as a selection constraint to restrict the data set. In each rule, $A \Rightarrow B$, $\text{set}(A) \subseteq \text{set}(B)$ because the rule "if A then B" means "all A's are B's".

The link *between successive rules* in a graph is a sub-set relationship since consequent_i 'matches' antecedent_{i+1} by subrange containment; both conditions refer to the same attribute and the set of values specified in consequent_i is a sub-set of those in antecedent_{i+1}. Adjacent-condition sets in a path may be the same size, (because consequent_i and antecedent_{i+1} match exactly, or the wider range antecedent selects no more tuples due to an empty subrange interval, or because A and B in a rule $A \Rightarrow B$ are equivalent conditions) but consecutive conditions in a path can never denote a *decrease* in set size.

Corollary 1: For any path, any reachable node whose attribute is the same as the start node represents a super-set of the start node set. For example, a path from node $(15 \le g \le 20)$ may lead to a node $(8 \le g \le 47)$ which is a super-range of the earlier condition and therefore denotes a super-set of tuples.

Corollary 2: String attribute assertions cannot lead to other assertions on that same attribute (unless the rules support set-valued assertions). This is because string attributes allow only equality or set-valued conditions (not ranges). A graph path starting with (a="BBC") could only lead to an assertion such as (a ∈{"BBC", "ITV"}) describing the properties of the reachable super-set.

Theorem 2: Reachable node assertions are non-contradictory

Proof: Starting from a specific node, such as (a= "BBC") any forward path in the directed graph denotes a monotonically increasing sequence of nested sets of tuples. When a path *branches*, the two alternative sub-sets at the branch point must have a non-empty intersection which contains the set selected by the original condition (a= "BBC"). This applies at every branch point reachable from that original condition. Therefore *all* the reachable nodes denote tuple sets which *overlap*. Each node assertion specifies sets of values found in the node's sub-set of tuples. *Contradictory assertions* have *disjoint value sets*. For example, one says $(2 \le g \le 10)$ and another $(15 \le b \le 20)$. This cannot occur with intersecting sets of tuples because the assertions must contain *common values*, found in tuples common to both sets.

Corollary 3: Alternative consequents can be intersected. Branching produces rules which involve alternative super-sets of the set selected by the antecedent. For example, using a(n .. m) as an abbreviation for $(n \le a \le m)$,

$a(5 .. 7) \Rightarrow b(11 .. 15)$ cascades with rules:

$b(10 .. 28) \Rightarrow c(7 .. 16)$ and $b(2 .. 16) \Rightarrow c(3 .. 10)$.

This produces two transitive rules:

$a(5 .. 7) \Rightarrow c(7 .. 16)$

and $a(5 .. 7) \Rightarrow c(3 .. 10)$.

The first rule states that any tuple with $(5 \le a \le 7)$ will have a c value between 7 and 16. The second asserts that their c value is somewhere between 3 and 10. Therefore:

$$a(5 .. 7) \Rightarrow c(7 .. 10).$$

This rule uses the intersection of the two consequents. The intersection must contain all the a(5 .. 7) tuples, and the narrower consequent range is a better description of that set than either of the consequents alone. Combining the knowledge from two different paths has produced a more specific consequent assertion about the subset selected by condition a(5 .. 7).

A narrow consequent range is desirable for SQO. So *graph pre-processing*, to intersect consequents from different branches, can give better query reformulation results than the traditional method which sequentially applies rules to the query. 'Better results' in this case means it can reformulate a wider range of queries using the given set of rules, since the traditional method may not use all alternative rules within its limited time.

A rule produced by intersecting consequents is a new edge in the Condition Dependency (CD) graph. Its narrowed consequent range can cascade with antecedents not previously used. These new transitive rules are derived during graph processing. They describe smaller sets than previously cascaded rules, so they narrow the intersection range of alternative consequents in the next stage of rule composition. (Wide antecedent ranges are associated with wide consequent ranges. Range intersection uses extreme values from those ranges. A narrow range provides *better* extreme values for the purpose of producing a narrow intersection range).

4 Properties of Paths

A rule A ⇒ K partitions the set of tuples in a database relation into three parts, as explained in Section 1. This causes a *Dilution Effect*, whereby the consequent assertion becomes progressively less specific to the antecedent set as set(K) becomes a progressively larger superset of set(A). This is a characteristic of paths in the graph. In any transitive rule, A ⇒ C, produced from:

$$(A \Rightarrow B) \wedge (B \Rightarrow C),$$

set(C) is a superset of set(B) and set(B) a superset of set(A); so set(C) may therefore be much larger than set(A).

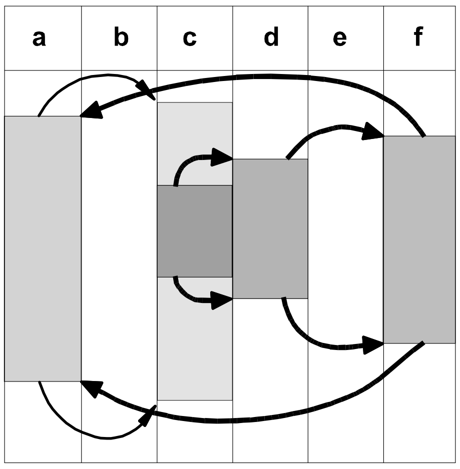
A rule, A ⇒ B, asserts that tuples which satisfy condition A will exhibit property B, but if a lot of other tuples also have property B the value of the assertion is limited. This affects query reformulation operations (listed in section 2.1). In operation 2.1.1 the consequent must be a subset of the implied query condition. Operations 2.1.2 and 2.1.3 require *non-intersection* with the consequent set. This becomes less likely as the set covers a larger fraction of the whole data set. Operations 2.1.7 and 2.1.8 require the added condition to be close in size to the existing antecedent condition to which it is being added in the query.

Because cascaded rules are less useful than direct rules it is probable that the best reformulated query will be found directly from the rule set rather than by taking longer in query reformulation to search the graph for transitive rules, or build them by successive application of rules to the query. This explains the experimental observation [Ti 97] that the reformulated query obtained quickly from a given set of rules is little improved by allowing unlimited time and using many more rules.

4.1 Maximum Usable Pathlength in the Graph

In order to reduce workload at query time it is desirable to pre-process the rule graph, to identify any *useful* transitive paths and add them as direct rules to the rule set. Rules discovered by this deductive method are no different in character from rules produced by data analysis. There is no risk of them becoming invalid if any rule in their chain of deduction was deleted, since they describe a connection between two columns in a database table, independent of the rules which revealed that connection. The graph pre-processor needs to know whether there is a maximum path length from any condition node, beyond which further exploration is unnecessary, when deriving these transitive rules.

Part of the answer is provided by the Dilution Effect, since the conditions in a path denote a sequence of monotonically increasing nested subsets of the database table. The two conditions in each rule are labelled with their percentage selectivity, so a path can be abandoned as soon as a new node makes the size difference between antecedent and consequent sets too large. But there is also a maximum path length independent of set sizes, as now explained.



A path is a sequence of conditions. Each condition refers to a single attribute. Successive conditions refer to different attributes. The sequence of rule consequents following a particular condition node in the chain are implied consequents of that condition. A chain of inference terminates when the *attribute* in the first condition is encountered again. Subsequent conditions in the path are consequents of that second condition on the antecedent attribute. They are also consequents of the first condition, of course, but such implied rules are logically redundant; subsumed by the later rule. For example, the following conditions occur in a path:

a(10 .. 15) → ... → a(5 .. 20) → ... → h(86 .. 117).

Two rules can be extracted:

a(10 .. 15) ⇒ h(86 .. 117),
a(5 .. 20) ⇒ h(86 .. 117).

The first rule is redundant according to the Rule Subsumption Theorem [Ro 97], since the second rule contains all the information in the first rule, and more, because it describes a super-set (*i.e.* those selected by the condition $(5 \le a \le 20)$) of the tuples with $(10 \le a \le 15)$. Therefore:

i) The maximum path-length that needs to be examined in a CD graph when deriving rules is N-1 edges (N nodes), where N is the number of columns in the database table and the path is a sequence of N conditions on the N different columns. Paths longer than N can *exist* in the graph, but rule derivation only needs to examine paths to depth N.

ii) As soon as a path gets back to the column from which it started, the node at the end of the path must denote a superset of the tuples identified by the start of path node condition. Eg: c(15..20) ⇒ c(12..93). So there are no further consequents in this path for the start node antecedent condition. c(12..93) starts a new path, providing information about a different subset of the database table. This is a super-set of the tuples described by the first path. A path does not necessarily pass through *all* other attributes before returning to the attribute in its start of path condition.

After graph pre-processing, each antecedent condition has a set of optimum consequent conditions associated with it. These are single-edged paths in the graph, which represent the discovered transitive and intersected-condition paths; but the set is more efficiently accessed if it is coalesced into a single rule for the single antecedent condition. A *vector* of consequents holds the set of rule consequents. For example, the rules:

$(a = 5) \Rightarrow (4 < b \leq 7)$ and $(a=5) \Rightarrow (d = \text{"bf235-k"})$

become: $(a = 5) \Rightarrow (4 < b \leq 7) \wedge (d = \text{"bf235-k"})$.

Each element in the consequent vector is a consequent condition from a rule. The vector is a conjunction of conditions, but since consequents are assertions rather than constraints, individual conditions can be used in the same way as the individual rules from which they were derived. Elements in the vector are ordered in the same way as the attributes to which they refer in the database table. Many attributes will be absent from a rule consequent, since only antecedent-consequent attribute pairs useful for query processing are included.

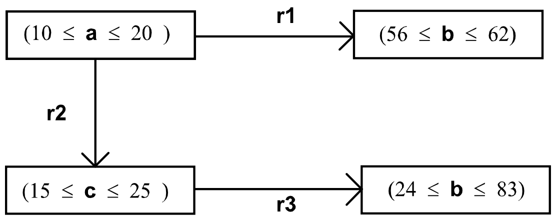
4.2 Inefficiency of Transitive Rules

A path is an inefficient way to derive a subset dependency rule between two attributes. Inefficient because the Dilution Effect produces *low utility* rules. Eg cascading the two rules:

r2: $(10 \leq a \leq 20) \Rightarrow (15 \leq c \leq 25)$ and
r3: $(14 \leq c \leq 32) \Rightarrow (24 \leq b \leq 83)$

provides a rule linking attributes a and b via c. The transitive rule is $(10 \leq a \leq 20) \Rightarrow (24 \leq b \leq 83)$.

But direct rule derivation, using the selection condition $(10 \leq a \leq 20)$ to extract a set of tuples and examining the range of values for attribute b in that set, will usually provide a narrower range, such as $(56 \leq b \leq 62)$.



Therefore it is questionable whether rules should be cascaded to provide transitive rules, because they are less useful than the corresponding direct rules. Although rule concatenation does provide information without accessing the data server, which may be beneficial in some environments.

If the rule set has not been pre-processed to discover transitive rules, the Dilution Effect suggests that transitive rules are *rarely* beneficial, so that *on average* the time saved by not looking for these rules *at query time* (and so delaying the query and adding to query response time) will contribute to an average improvement in query response time. The rare occasions when a useful transitive rule is discovered will not outweigh the time wasted on other queries.

4.3 Search Space Reduction

If the depth of search in the Graph *at query time* is limited to one edge, then a query with N conditions requires N rule antecedent lookups. This must provide (i) paths between query conditions, (ii) paths to conflicting consequents, and (iii) paths to low cost conditions which can be redundantly added to the query, as indexed tuple-retrieval conditions or filter conditions.

As well as reducing the number of sequential paths to consider at query time, it is also important to provide rapid simultaneous access to parallel paths because each query condition on an N-ary database relation can have up to N-1 outgoing edges from it, for each of the other attributes in consequent conditions. Multi-consequent rules, introduced in Section 6, provide this support.

The set selected by an end of path condition is larger than the set selected by the start of path condition. So unless the relative selectivity of two *query conditions* (estimated from attribute histograms) is appropriate it is not worth searching for *any* path between them.

The Dilution Effect also suggests that (in a reformulated query) an added index or filter condition obtained from a transitive path will be a poor selector of relevant tuples for the existing query condition which is the start node in the transitive path. Selecting a significantly larger set is not useful and can increase the time needed to process a query.

5 Potential Rulebase Size

A common problem for systems which use automatic rule discovery is the tendency of the rulebase to grow unsuitably large. For the type of rules used in SQO, each N-ary database table can spawn up to N * (N-1) different *sets* of AP rules, plus N * (N-1) * (N-2) *sets* of 2-antecedent-condition rules. In general, for rules with exactly P antecedent conditions there are N!/(N-(P+1))! different *sets* of rules, reflecting different combinations of attributes on the left hand side of rules. Any of these numerous sets of rules can grow very large, e.g. for an integer type antecedent attribute with m different values appearing in the attribute column of the database table there are ${}^{m}C_2$ different ordered pairs of integer values to use as range specifiers, i.e. m*(m-1)/2 possible range intervals to use in antecedent conditions. For example, if 2000 tuples contain 1000 different values for this integer attribute there are 499500 possible rules with this attribute as antecedent. And this is only one of the N * (N-1) rule sets which use only a single antecedent condition.

It is easy to see how a rulebase can become many orders of magnitude larger than the data it describes, unless it is deliberately restrained. Using histograms as the basis of a rule sets, as described in the following section, is a restraint mechanism. The number of rules in the set for any particular antecedent attribute is now chosen in advance, before deriving rules from the data.

6 Histogram for Antecedent Conditions

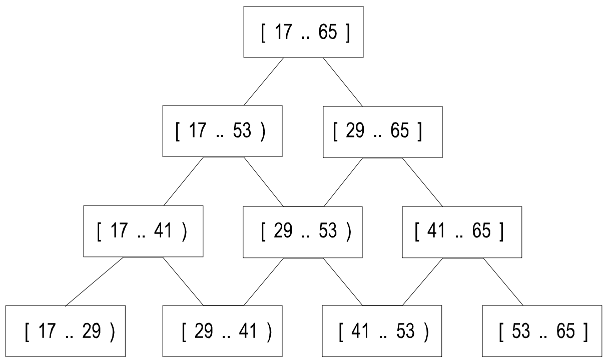
If the rule set for a numerical antecedent attribute is allowed to grow automatically, by rule discovery, then the very large set of range rules produced contains much redundancy. Many rules are subsumed by others, and there is much nesting of closely-related sets of tuples defined by rule antecedents [Ro 97]. The redundant rules can be removed, but it is inefficient to generate a large set of rules and then analyse and delete most of them. Better to decide the size of the required rule set in advance. Then derive antecedent condition ranges by dividing the interval between the two extreme values (highest and lowest for the attribute) by the number of rules. Rules therefore correspond to bars in a histogram for the antecedent attribute. Histograms are already used in *conventional* query optimisation (distinct from SQO) to estimate the size of intermediate tuple sets. A single (multi-consequent) subset descriptor rule is now attached to each bar of the histogram. This provides multiple assertions per bar, to supplement the single, frequency value, assertion. An N-bar histogram limits the size of the rule set per antecedent attribute to N rules.

Category or string attributes can be dealt with in the same way. *E.g.* 100 bars representing 100 category values with bars taller than a certain frequency-of-occurrence value. Each rule is known to describe a suitably significant percentage of tuples in the table.

Examples of rule antecedent conditions associated with bars are: (job_title = 'secretary'),
or (7.4.65 ≤ date_of_birth ≤ 7.4.75).

A particular merit of the histogram rule set for a numeric antecedent attribute is that it represents a *hierarchy of other rules*, quickly derivable from the base set. Adjacent bars in the histogram can be combined to provide more rules. The antecedent range can therefore expand to capture any 'comparable' query condition, by concatenating adjacent rule ranges. This provides a solution to the query condition containment problem suffered by other rule sets, where a large number of rules is needed in order to find one to contain the query condition range fairly closely. Any number of adjacent rules can be combined, from 2 to 99 rules in the case of 100 ranges. For N original rules, N-1 new rules can be formed from adjacent pairs, N-2 further rules from adjacent triplets, N-3 rules from groups of four adjacent rules, etc. Hence N + (N-1) + (N-2) + ... + 3 + 2 + 1 = N * (N+1)/2 rules are represented by a basic set of N rules, *i.e.* half the square of the number of basic rules, approximately. (About 5000 in the case of 100 rules. But rule maintenance is much easier with only 100 rules).

The following diagram shows the hierarchy of antecedent ranges available from a base set of four rules. The two extreme values in the antecedent attribute are 17 and 65. This attribute range [17 .. 65] was divided into four sub-ranges to produce the histogram rule set:



The antecedent ranges for the corresponding four rules are shown in the lowest row. But these four rules represent 10 rules, by Unioning. A Union of Antecedent conditions implies a union of rule Consequent assertions.

Unioning is a rapid process because of the simple rule structure. *The purpose* of unioning at query time is to produce a rule antecedent range that brackets a query condition, and is therefore implied by the query condition. Eg: a query condition range [34 .. 42] on the attribute to which these antecedent ranges refer, uses the antecedent [29 .. 53). This rule describes a set of tuples which includes all the items of interest to the query. Any further conditions in the conjunctive query operate to restrict the set selected by the query condition [34 .. 42] on this attribute.

Brackets in the diagram follow the usual convention for interval representation, that a square bracket means the limit value is included in the range, while a round bracket indicates the limit value is not included. E.g. [29 .. 53) = (29 ≤ value < 53).

6.1 Multi-Consequent Rules

Rules are derived from data by using the antecedent condition to select a set of tuples and then producing consequents which describe the tuples in that set. To support fast query reformulation and rule maintenance the histogram for a particular antecedent attribute is used to provide antecedent conditions in *multi-consequent rules*. Each of these rules is a compact representation of a set of (up to N-1 for an N-ary relation) AP rules which all share the same antecedent condition but differ in their consequent attribute. Each multi-consequent rule can therefore provide information about the values of all attributes in the subset of tuples selected by the antecedent condition.

The multiple consequents are ordered according to the position of their attributes in the relation schema, so that two or more of these consequent *structures* can be rapidly compared or their field values unioned. Comparisons are useful between consequents implied by different query conditions; and unioning is useful when concatenating antecedent ranges to produce a range wide enough to enclose a query condition.

A multi-consequent rule $C \Rightarrow A \wedge B \wedge D \wedge E$ (where A .. E are conditions, ie atomic constraints, on attributes a .. e respectively) has a *conjunctive* consequent assertion. The consequent is a logical product, so all its component assertions are true. Therefore any of those components can be used or ignored according to the needs of the rule application.

A multi-consequent rule set is a TABLE whose column names are the same as those of the database relation it describes; *i.e.* column names are attribute names, and rows correspond to bars in the histogram describing a particular attribute currently used as antecedent for rules. Each row in the table is labelled in the same way as a bar in a histogram, to identify the sub-set of tuples it describes. Eg (job_title = "nurse") or ($53 \le f \le 65$). But the whole table refers to a single antecedent attribute, so there is no need to mention the attribute name (*eg* job_title, or f) in row labels. The table structure provides *fast lookup* by row label:

	a	b	c	d	e
[17..29)					
[29..41)					
[41..53)					
[53..65)					

Different antecedent attributes have different tables, but, since the *structure of rows* is the same in all rule tables for a particular database relation, they are easily compared. Each row is a vector of attribute assertions, so information about a particular attribute will be found in its fixed position slot in the row. Contradictory consequents, conflicting query condition requirements, single values for result attributes, etc., are thus directly accessible. The particular rows to compare are chosen using the row label that 'matches' a query condition.

Example 1: a query on a personnel database concerns people whose (job_title = "nurse") AND (salary < 12 K).

Using the relevant multi-consequent rule table, whose antecedent attribute is 'job_title'; if the row labelled "nurse" shows an entry such as [12741 .. 19380] for the 'salary' column there are no tuples matching both query conditions. Furthermore, the system can easily provide an *explanation* for its answer, e.g. "There are no result values *because* (job_title = "nurse") implies salary is in the range £12741 to £19380, in this data set".

Example 2: *contradictory consequents*.

A new query contains two selection conditions:

$$(e = \text{"nurse"}) \wedge (50 \le g \le 100).$$

Each condition restricts a different attribute, so the rule sets for attributes 'e' and 'g' are consulted. The following rules 'match' (i.e. their antecedents are implied by) the two query conditions.

(e = "nurse") ⇒ a(15..20), b(106..183), c ="TV", d ≥ 101
(50 ≤g≤ 100)⇒ a(18..34), b(44..71), -- , d(80..110)

Consequent values above each other are easily compared, to implement operation 2.1.3, and the ordered format of consequent assertions facilitates operation 2.1.2.

Contradictory assertions about the value of attribute 'b' show that the two query conditions (e = "nurse") and ($50 \le g \le 100$) are incompatible. They denote disjoint sub-sets of the data, so the query will return no result tuples.

7 Discussion

Section 3 identified properties of a rule set when viewed as a condition dependency graph. This clarified the requirements of semantic query reformulation operations, and showed that time-consuming operations (which counteract the effectiveness of query optimisation) can be done independently of the query, as a pre-processing phase. Special-purpose rules extend the expressiveness of attribute-pair rules, and these link into the CD graph via their single consequent assertion per attribute. The antecedent can be seen as a constraint on a sub-relation or view, but the rule's consequent assertion applies to tuples in the parent table described by the graph.

The result of graph pre-processing is a vector of attribute value assertions attached to each antecedent condition in the graph rule set. If the set of antecedents for a particular attribute were assembled, each with its vector of consequents, it would resemble a multi-consequent histogram, but the antecedent ranges in *this* set can overlap and can be nested. Nevertheless, this set of antecedent ranges is an ordered set, and could therefore be organised in an appropriate data structure for fast lookup. This is explained further in [Ro 98]. That paper also discusses criteria to identify useful sub-sets in order to restrict the number of rules. (Since each antecedent condition is a sub-set selector for tuples in a virtual or base relation in the database).

Section 5 explained that the worst case size for a set of rules with only one antecedent condition on a single attribute is m * (m-1)/2 rules, where m is be number of tuples in the database relation (all with different values, in the worst case). If all N attributes from the relation were used as antecedent conditions the total could grow to N*m*(m-1)/2 rules. If two antecedent conditions per rule are allowed, the rule set produced by automatic knowledge discovery could grow by a *further* (N*m*(m-1)/2) * ((N-1)*m*(m-1)/2) rules. Therefore section 6 suggested the set partitioning strategy used by histograms as a direct source of rule antecedents, rather than unguided discovery.

Histogram rule sets provide, directly, a *single* rule matching each query condition. This is another advantage of histogram rules versus rule sets with overlapping antecedent ranges, since those require the extra work of deciding which of the applicable rules to use.

The histogram rule set is an automatically derived data profile whose easily understandable structure allows it to be displayed to users wishing to understand the data. Other rule bases, in contrast, can be incomprehensible to a user who wants an overview of data characteristics. The histogram can also be used to provide *explanations* for empty query result sets, and to provide *summary answers* (specifying the range for each attribute in the result set) instead of tuple sets. Another way to summarise the result set is by showing a semantically equivalent *query*, since this *describes* the query-specified data set in a different way, which may be informative.

The knowledge representation scheme proposed in this paper also allows aggregate values to be pre-computed for the sub-set of tuples described by each rule, and used to further accelerate query response. Support for aggregates is less viable in the amorphous rule sets commonly used.

8 Conclusions

Semantic optimisation is a strictly time-constrained task, so the central issue for effective implementation is how to make best use of the limited time available. Elaborate or unstructured rule sets are unlikely to be accessible in time. Sets of attribute-pair rules (and their multi-consequent representations) are easily structured for rapid access. The Rule graph provides a map of possible query reformulation operations, which allows (i) appropriate rules (paths) to be directly selected, and (ii) alternative and transitive paths to be precomputed and assessed before query time. (Full transitive closure is not required, since the maximum useful path length for an N-ary base relation is N nodes).

An explanation was needed for the empirical observation that the best query reformulation is usually produced quickly from the given rule base, before many of the query-relevant rules have been used. The Dilution Effect suggests that rule quality is inversely proportional to inference path length for that rule. Therefore applying rules to the results of previous rule applications will probably not be beneficial. Furthermore, the empirical result suggested that early termination of query processing was desirable but it was not clear when to stop. Limiting graph search to a maximum pathlength of one edge makes termination time explicit. Graph preprocesing can convert any useful transitive paths to single-edge paths in preparation for this fast query reformulation strategy.

The *set* of reachable nodes at distance 1 from each query node must then be retrieved during query processing. Multi-consequent rules were proposed as a way to add value to each rule antecedent lookup. Each of these rules effectively explores all *alternative* paths from a graph node such as a query condition, and provides structured results whose components are easily compared or combined with results from other lookups. These histogram-based rule sets also solve the problems of large set size and difficult rule maintenance, which are characteristic of incrementally derived rule sets.

A multi-consequent rule set can be produced by a single scan through the database table, since it involves assigning tuples to buckets rather than sorting. Continuous incremental aggregate value production for each consequent condition in each bucket allows very large tables to be condensed to a rule set during the scan. The work is easily distributable to multiple processors or workstations.

The effects of cascading rules, which are revealed by the graph, also apply to successive application of rules of any structure. So the conclusions in this paper have general application to SQO, not just to attribute pair rules.

References

[Hs 95] C. N. Hsu, C.A. Knoblock, *Using Inductive Learning to Generate Rules for Semantic Query Optimization*, Chapter 17 in *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1995.

[Ki 81] J.J. King, *QUIST: A System for Semantic Query Optimization in Relational Database Management Systems*, Proc. VLDB Conference 1981

[Lo 95] B.G.T. Lowden, J. Robinson, K.Y. Lim, *A Semantic Query Optimiser Using Automatic Rule Derivation*, Proc. WITS '95, 5th International Workshop on Information Technologies and Systems 1995, pp 68-76.

[Ro 97] J. Robinson, B.G.T. Lowden, *Data Analysis for Query Processing*, Proc IDA '97, Second Intl. Symposium on Intelligent Data Analysis, 1997, pp 447-458 (LNCS 1280).

[Ro 98] J. Robinson and B.G.T. Lowden, *Query Optimisation using Subset Descriptors*, submitted for publication, May 1998.

[Sh 88] S. Shekhar, J. Srivastava, S. Dutta, *A Formal Model of Trade-off Between Optimization and Execution Costs in Semantic Query Optimization*, Proc 14th VLDB Conference 1988, pp 457-467.

[Sh 87] S.T. Shenoy, Z.M. Ozsoyoglu, *A System for Semantic Query Optimization*, Proc ACM SIGMOD Conference, 1987, pp 181-195.

[Sh 88] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle. *Learning transformation rules for semantic query optimization: A data-driven approach,* IEEE Transactions on Knowledge and Data Engineering, 5(6), 1993, pp 950-964.

[Si 88] M. D. Siegel, *Automatic Rule Derivation for Semantic Query Optimization*, Proc 2nd Intl Conf on Expert Database Systems, 1988, pp 371-385.

[Si 91] M. Siegel, E. Sciore, S. Salveter, *Rule Discovery for Query Optimization,* in *Knowledge Discovery in Databases,* AAAI/MIT Press, 1991, pp 411-427.

[Ti 97] K. Tint, *Semantic Query Optimisation,* MSc Project Dissertation September 1997, Department of Computer Science, University of Essex.