Data Analysis for Query Processing

J. Robinson, University of Essex

Abstract. Data analysis is needed in connection with query processing, to produce data summary information in the form of rules or assertions that allow semantic query optimisation or direct query answering without consulting the data itself. The goal of an intelligent analyser in this context is to produce robust rules, stable in the presence of data changes, which allow easy rule maintenance as data changes, and provide rapid query reformulation, refutation or answering. It must also limit the rule set to rules useful for query processing.

1. Introduction

The purpose of the data analysis considered in this paper is to improve the speed of answering queries on that data. Analysis produces summary information that can either answer a query without consulting the data itself, or else modify the query to a form the data server will be able to process more quickly. This query-modification operation using knowledge of the data is known as Semantic Query Optimisation (SQO) [1, 3, 5,7]. *Relational* data is discussed in this paper.

This application requires *continuous data analysis*, as the focus of query interest in the database changes with time. The analyser's activity is guided partly by information it discovers during examination of the data, and partly from query information such as query structure, history of data access, and the frequency of data change in certain areas of the database.

Continuous background data analyser processes (which can utilise idle time on any available networked workstations) derive and shortlist useful rules (summary information) for the query reformulation module, and receive copies of queries and data updates. The query processor module can recognise any summary information affected by data changes, because of its simple format, and suspend it from use. Although fast data analysis is desirable there is no critical dependency: any current query will utilise information available at the time, but there is no delay, waiting for analysis results. Because of this, the continuous analysis workload can be distributed to any available workstations in the local network, to run as background processes. However, the amount of spare computing capacity in a modern workstation is high even when the machine is 'in use' rather than 'screen saving', so analysis results tend to be available in time to be used for the next query.

Previous work in connection with meta-data discovery for use in query processing has used a machine-learning approach where the result of a query is treated as a set of positive instances (a training set) [1, 3]. A disadvantage of this approach is that it can produce exactly the rules that are unlikely to be used by future queries, unless the same query is repeated in the near future. The reason for using query-triggering of rule induction was a fear of numbers of discoverable rules. However, queries can be used by an intelligent data analyser to suggest beneficial directions for analysis without unnecessary restrictions.

This paper examines the potential size of discoverable knowledge and discusses ways to impose beneficial and appropriate restrictions. The products of data analysis are *Assertions* and *Inference Rules*, as explained in section 3.

The structure of paper is as follows: section 2 introduces the application requiring this form of data analysis, section 3 discusses the type of rules to be produced by the analyser to support the application, and discusses characteristics of these rules and decisions the analyser must make during the analysis process and based on its observations of data features.

2. Query Reformulation

An example of a Database Query or subquery is:

"Obtain column6 values from rows of database table J, where (column2 = "AB6") AND (column4 is BETWEEN 15 AND 46)."

The bracketed components are the query *Conditions*. Each is a Selection Condition, denoting a subset of the database table (ie the subset of tuples matching that condition). Conditions, or *Constraints*, have the form $(a\ \theta\ n)$ or $(n \le a \le m)$ where a is an attribute name and n, m are constants of the same type as attribute a. θ is an operator from the set $\{ <, \le, =, \ge, > \}$.

Condition matching between queries and rules is needed, to reformulate the query. Matching is by *subset containment*, as explained in section 3.3 below. This is also the mechanism for cascading rules to form the CD Graph (section 3.1). Containment mapping means that rule conditions can be used without necessarily matching a query condition exactly - which would limit the usefulness of a rule. One measure of the usefulness, or Utility, of a rule is the range of different query conditions it can match. Section 3.3 identifies *Containment Potential* of Conditions as one of the factors an intelligent data analyser uses to guide its rule derivation operations.

The Query Rewrite Module uses information produced by data analysis to either answer or rewrite the query. The requirements of these operations dictate the form of information to be obtained by data analysis.

(i) answer a query immediately by

a) reporting that there are no values in the result set, or

b) returning the single-value of the query-specified attribute which is provided by the consequent of a rule, or

c) in specialised systems, providing an intensional answer rather than a set of data values.

(ii) modify the query before sending it on to the data server to process in the normal way (using conventional query optimisation to generate an execution plan). The modified query will provide a faster execution plan. *Query reformulation methods* include the following *seven operations*.

1. Query Condition Deletion

Either *(a)* because the deleted condition's truth is implied by another query condition, which will be tested during query processing in the data server. So there is no need to test both conditions. Any tuple that satisfies the first condition is known (by previous data analysis) to satisfy the second.

or *(b)* if the selection condition in the query selects all tuples in the database table. Its range contains both *extreme values* for the attribute. The condition is therefore true of all tuples in the table, so need not be tested during query processing.

2. Query Condition Substitution
by an equivalent but faster-to-test condition. (Equivalent conditions denote the same set of rows in a database table).

3. Index Introduction
Adding one new condition to a query which contains no *indexed attribute* conditions. The new selection condition extracts a query-relevant subset of rows from the table, to avoid scanning the whole table, applying the other query conditions to each tuple. Therefore the intelligent data analyser must generate rules A => B where B is a condition on an indexed attribute and B selects a small superset of the tuples identified by condition A.

4. Query Refutation
Two ANDed query conditions A and B are incompatible if they denote different subsets of the data. The query will produce no results. To support this, the data analyser must have produced rule A => C where the attribute in C and B is the same, but the two condition ranges are disjoint.

5. Query Answering
Query: "Get the values in the d column from all rows of a specified database table where the value in the e column is in the range (52..63)".
Rule: `e(25..90)  =>  (d = "secretary").`
The rule says: If a tuple has an e value in the range (25..90) its d value is "secretary". So return the single result value, "secretary", in answer to the query.

6. Join Elimination
From semi-join queries (where result values come from only one of the joined tables) the Join operation can be avoided if the data analyser has produced rules in advance from the appropriate joined table, which show the query condition on the second table is always true if one of the query conditions applied to the first table is true. The data analyser uses information from previous queries to decide *which tables to join*, *which join attributes to use*, and *which pair of attributes* in the joined table to use for antecedent and consequent conditions in its derived rule set. This restricts rule derivation to rules known to be relevant to queries.

7. Scan Reduction
In a Join query, if one of the tables has no selection condition then the whole table must be scanned repeatedly in comparing the join attribute values of two tables. However, a new selection condition added to the query can make the data server select a subset of the table before the scan phase. A condition can be added if it is implied by an existing query condition. This implication will have been noted, in the form of a rule, by appropriate data analysis for this purpose. The data analyser is aware that high selectivity is required in the consequent condition of rules it derives. It also

knows that the consequent condition always selects a superset of the antecedent condition's set, so its systematic data analysis will terminate when either antecedent or consequent denote too large a percentage of the table. But since tuple numbers rather than percentages are the basis of cost, it will also consider the size of the table when choosing the point at which to terminate analysis.

3. Data Analysis to Support Query Processing

Two sorts of information result from data analysis: *Assertions* and *Inference Rules.* Assertions are sets of statements about a table or attribute. For example, D*omain Assertions* for an attribute include: the two *extreme values;* a set of *empty subrange* intervals on the number line between the two extreme values, each denoted by a pair of range limits; *percentile points* are useful during rule analysis to estimate the selectivity of range conditions, but may not be maintained afterwards. The *number of distinct* values in each column of the table, and the *set of distinct values* for category attributes, are examples of other domain assertions.

Inference rules are pairs of *conditions*, each on a different attribute in the table. For Join Rules the table is the one resulting from the Join operation, and each attribute in the rule is from a different table. Rule characteristics are discussed in section 3.2, but one of their main features is that each rule is an *edge* in a graph.

3.1 The Condition Dependency Graph

Rules can be cascaded. For example, in the three rules:

```
A => b(4..15)        b(3..20) => C          b(2..32) => D
```

where A, C, D and b(n .. m) are conditions. The consequent condition (assertion) b(4..15) satisfies antecedent condition b(3..20), so `A => C`. Similarly, by transitive inference, `A => D` using the first and third rules.

The *rule of inference* that allows the consequent of one rule to imply the antecedent of another rule is *subset containment* since a Condition denotes a set of tuples. In the case of *range* conditions, implication is also by *subrange containment.*

Several rules can be cascaded to form a chain or branched path. Rules are therefore edges in a Condition Dependency Graph, whose paths are chains of inference leading from a given Condition (such as a selection condition in a query) to another Condition, an assertion that restricts possible values of a specific attribute.

Semantic Query Optimisation can thus be seen as path discovery in the Condition Dependency Graph. Query conditions map to condition nodes in the graph, where paths reveal redundant query conditions, available conditions on indexed attributes that can be added to the query, lower cost Equivalent Conditions (denoting the same subset of data items), and in some cases paths to restriction conditions that immediately refute the query or provide a single-value answer to the query.

Fig 1 shows that condition E can be deleted from the query. The path from A to E means that testing condition A is sufficient to test both conditions A and E. The rule A => E denoted by this path will have been derived by the data analyser in advance of query processing, so there is no graph traversal at query time, only table lookup in the condition-pair rules for pairs of conditions in the query. Rules the query reformulator seeks initially, to see whether any conditions can be deleted from the query, are: A => F, A => E, F => A, F => E, E => A, E => F. Conditions F, G, H and I are *equivalent conditions*. They all denote the same set of tuples, so G, H or I could be substituted for F in the query. If rule F => H gives best cost reduction substitute H for F in the query. The data analyser has derived rules F <=> G, F <=> H, F <=> I, tagged as *equivalences* and labelled with degree of cost reduction between conditions measured as in [3]. If the query still lacks a condition on an indexed attribute and L is an indexed condition with suitable selectivity it can be added to the query because of the path (ie rule) A => L.

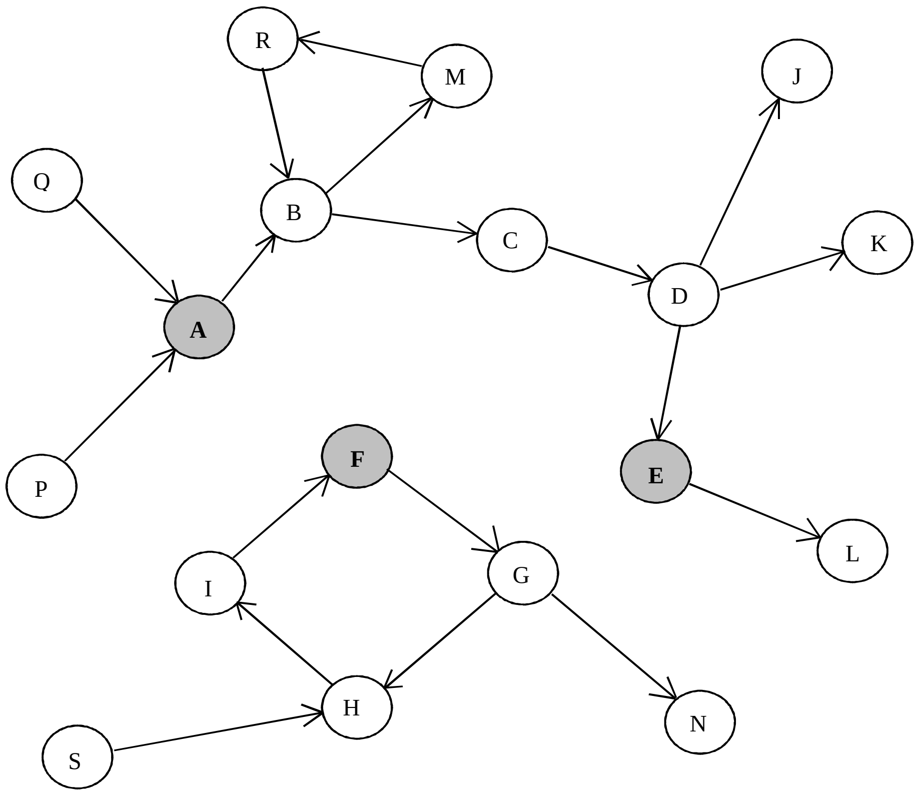

Fig 1: A Condition Dependency Graph including query conditions A, F and E.

3.2 Rule Discovery by Data Analysis

Attribute-pair rules, called 'Simple Rules' in [7], are particularly suitable for query processing. The single condition structure of antecedent and consequent optimises rule usability, (*all* conditions in a rule must be matched). The *robustness* [2] of a rule is the probability that the next data change will not affect it. Attribute-pair rules have only two attributes to be affected, so their robustness is correspondingly higher than rules with two antecedent conditions, for example.

Other advantages of the 'simple' rule structure are easy rule extraction from data, fast access (for query processing and rule maintenance), and the ability to precompute all chains of inference that can occur with a query .. avoiding delay during query processing. *Fig 1* shows query reformulation is modification to *improve* the cost of the existing query. Relative cost of the two conditions in certain attribute-pair rules is examined by the data analyser. Operations 2 and 3 in section 2 involve relative cost, the rest do not. So only indexed-consequent rules and rules in a cycle of the CD graph need condition cost assessment by the data analyser.

Attribute Types and Conditions

Standard SQL data types include numeric types for which ranges can be specified and comparisons other than equality can be used in Conditions. String types, on the other hand, have equality and Set conditions, such as (job_title = "secretary") or (job_title IN {"doctor", "lawyer", "academic"}. Range or magnitude-comparison conditions are unlikely in queries on these attributes. String types are often used for categories, such as job titles, so the number of different values occurring in that column of the table is limited. The data analyser derives rules to operate on anticipated queries, so rules with string antecedent attributes have *equality* conditions; numeric conditions use *equality, magnitude comparison*, and *range membership*.

Attribute-Range Antecedent Rules

There are three forms of antecedent condition: $(a \le n)$, $(a \ge n)$, and $(n \le a \le m)$.

The third form is most useful in query processing, but the rule set is largest. The number of possible rules with antecedent $(a \le n)$ is N-1, where N is the number of different values that occur in the 'a' column of the table. Similarly there would be N-1 rules with antecedent $(a \ge n)$ for that attribute. But the number of possible antecedents of type $(n \le a \le m)$ is the number of pairs of values that can be chosen from the N different values, to produce pairs of range limits. The number is ${}^{N}C_{2}$. So there are N*(N-1)/2 different ranges. This is the maximum size of the ruleset for this attribute. *Eg:* for N = 100 there could be up to 4950 rules.

The value N is not necessarily related to the number of *tuples* in the table, but there is still a clear possibility that the rule set can be much larger than the database table it describes. (And this is only one ruleset for one antecedent attribute).

For each numeric type attribute, there can be *N rules with antecedents of the form (a = n),* plus *N-1 rules with antecedents of the form* $(a \le n)$, plus *N-1 rules with antecedents* $(a \ge n)$, plus *up to N*(N-1)/2 rules with antecedent form* $(n \le a \le m)$. The first three sets can be part of the fourth set, but for small tables it is desirable to keep *equality antecedent* rules as a separate set, and for large value sets delete them.

The numbers of possible rules will obviously increase if rules with more than one antecedent condition are permitted. The analyser therefore has to decide which rules should be derived, and which should then be retained as query requirements change with time. Ways to limit the size of range rule sets, and to decide the usefulness of rules, are discussed below.

3.3 Utility Measures for Rules

In a single-table rule, each condition denotes a subset of tuples in the table. Because A => B, set A is a subset of set B, so constraint (selection condition) B subsumes A. Because of this subset relationship, a rule consequent condition on an indexed

attribute may be added to a query if it will act as a quick source of relevant tuples from the table. But it should *(i)* provide a small superset of the tuples needed by existing conditions in the query, and *(ii)* denote a ‘small’ percentage of the table, since the cost reduction of using an index is the reduction in tuples to be tested, compared with scanning the whole table. (The analyser will take account of table size in deciding what percentage to regard as ‘small’, since cost reduction actually depends on *number of tuples* eliminated).
The rule set is reduced by discarding rules which fail to meet the required ‘utility level’. Condition selectivity can be estimated by the analyser from *percentile points* in the *domain assertions* set for antecedent and consequent respectively. Percentage is sufficient because A is a subset of B, so the percentage of the table denoted by A is part of the percentage denoted by B. Percentile information plays a different role in utility assessment if A => B is used for *deleting* B from a query containing both A and B, namely *containment potential*, discussed in the next section.

A path in the condition dependency graph denotes a sequence of monotonically increasing sets of tuples. A path is a succession of gradually increasing sized subsets of the database table. Consecutive subsets are the same size if the corresponding successive conditions in the path are Equivalent Conditions.

3.4 Condition Matching, Containment Mapping

The *Containment Potential* of Conditions is important because subrange containment or set containment are used to match Conditions, ie to use rules. It can be quantified as a pair of numbers :
a) condition range as a percentage of extreme value range for the attribute,
b) number of tuples selected by the condition as percentage of number in whole table.

The data analyser uses this measure to judge potential use for new rules. In the case of Join rules, the percentage refers to the *joined table* from which the attribute-pair rules are extracted. A good rule, `A=>B`, is one where the containment potential does not decrease much from A to B.

3.5 Range Antecedent Rule Production

A scanning algorithm can be used in systematic data analysis, to discover rules with interval range antecedent conditions. If the antecedent attribute is sorted into ascending order, its discrete values become apparent and the set of possible ranges is readily extracted. Eg if the sorted values were `1, 2, 3, 3, 5, 5, 5, 7, 9` then the ranges identified by a systematic scanning algorithm are:

```
1..2,  1..3,  1..5,  1..7,  1..9
       2..3,  2..5,  2..7,  2..9
              3..5,  3..7,  3..9
                     5..7,  5..9
                            7..9
```

Each range corresponds to an antecedent condition in a rule. The corresponding range of data values for the consequent constraint is obtained from the tuples selected by the antecedent condition.

After sorting the table on the antecedent attribute, the scanning algorithm uses two pointers to generate range antecedents and corresponding attribute-pair rules.

An example illustrates the process:

```
  a     b    c
->1     6    12
  2     4    4
->2     18   12
  3     9    4                         a(1..2) => b(4..18)
  5     3    12
  5     7    4                         a(1..2) => c(4..12)
  5     1    4
  7     26   5
  9     28   12
```

The upper-limit range pointer advances to find a data value different from that indicated by the lower limit pointer. Then advances to the last occurrence of that data value in the sorted sequence. A range of tuples now lies between the two pointers. These tuples are the set selected by the range constraint. The analyser simply has to identify the extreme values of b and c in this set to generate two rules shown to the right of the table. The upper limit pointer now advances to the final copy of the next value of a *and notes the new tuples it passes*. In the example it advances to 3 and adds only one tuple to its *difference set.* Values in the difference set are examined to see whether they extend the range of the previous rule consequent. b = 9 which is within the existing consequent range b(4..18). The new rule
`a(1..3) => b(4..18)`*subsumes* the previous rule: `a(1..2) => b(4..18)` which can therefore be deleted immediately from the rule set. The remaining rule is more useful than the deleted rule, and no information is lost by the deletion. It was an example of a logically redundant rule. If retained it degrades system performance by slowing access to the rule set and increasing rule maintenance overhead.

Rule Subsumption Theorem
For two rules, R1: A =>B, and R2: C => D,
R1 subsumes R2 if the ranges specified in conditions are $A \geq C$ *AND* $B \leq D$.
The proof is as follows:

- If $A \geq C$, eg a(5..100) > a(25..90), then C *implies* A. If C is true then A is true. Therefore C => A *and* A =>B so C => B
- now C => B and C => D but B is a more restrictive range condition than D. For example, compare the rules: C => e(25..50) and C => e(13..100). The first rule implies the second, by *subrange containment*: A value in the range (n .. m) is also in *any* range (≤n .. ≥m).

The data-scanning algorithm continues on the example table above, advancing the range upper limit pointer and noting whether attribute values in tuples in the difference set extend the existing consequent range or not, and deleting the previous rule if not. For the c column, the only rule remaining at the end of this first scan is `a(1..9) => c(4..12).` And this will be deleted because both conditions denote the whole table, so the rule has no information content. The data analyser uses the *extreme values* domain assertion for attributes, to recognise these redundant conditions and discard the corresponding rules.

The scanning algorithm now continues in the obvious way: the lower limit pointer advances to the *first instance* of the next distinct value of the sorted antecedent attribute and the upper limit pointer searches forward from this value to the *last instance* of the next distinct data value. This initialises the range for the next scan, and consequent conditions are computed from the initial set enclosed by the limit pointers. A useful feature of the scanning algorithm is the way its local *zone of interest* moves progressively downwards through the table during each scan. The zone is likely to fit in a memory page so the algorithm is well-suited to the virtual memory system of workstations. (Suitable for networked clusters, eg using PVM).

3.6 Path Extension Rule Production

The next phase of data analysis takes the set of rules produced by scanning and uses Antecedent and Consequent Triggering as a rule discovery algorithm. This extends CD Graph paths associated with rules (edges) known to be useful, and so adds value to those rules. Examples of rules known to be useful are indexed consequent rules (including Join rules) and Join rules with closely matching selectivity in antecedent and consequent conditions, so these are derived first by scanning. Consequent a(m..n) requires new rules with antecedent a(≤n..≥m) for path extension. Data examination reveals consequents in other attributes for these antecedent ranges. From a rule A => B try first to produce a rule B => A showing conditions A and B are *equivalent*, then generate a set of rules B => C, and hence infer rules A => C. This transitive deduction of attribute-pair rules from a CD graph path generates new rules that are independent of the rules used to reveal them. In a chain of inference corresponding to a path in the graph, intermediate rules can be eliminated by changes to data but the derived rule need not be affected. It is a rule linking values in two attributes, and only chages to those data values affect the rules.

Example: extend the graph from rule `d(51..90) => f(16..36).`
Use condition f(16..36) to select tuples from the table. The range for each other attribute gives a consequent C in a rule f(16..36) => C.

Selection condition d(51..90) derives positive instances. The negative set is selected by ((d < 51) OR (d > 90)). For each target antecedent attribute gradually narrow the range in the new rule's consequent condition, d(≥51.. ≤90), to eliminate all negatives from the positive set of antecedent values. Hence antecedent ranges.

4. Discussion and Conclusions

Early work in SQO used Integrity Constraints as the rule base. This provided *immutable* rules, immune to data changes, but needed a person to specify the knowledge, and integrity rules are not well suited to SQO requirements. More recent work has concentrated on automatic discovery of rules. We [3] and Hsu [1] used machine learning to generate rules that would have been useful if they had existed

before the query that triggered their discovery. Siegel [7] used queries in a similar way, to suggest rules that would have been useful, but examines the data directly to test whether the rules are supported by the data. We now *generalise* the query-recommended rule to a pattern for a *set* of rules to be obtained by data analysis, which then examines and shortlists the most probably useful subset for future queries.
Shekhar [5] used a classification grid to examine the distribution of data values and derive rules with a wide range of structures, whose applicability to a wide range of queries, robustness against data changes, ease of maintenance, and access speed is poor compared with attribute-pair rules.

Our data analysis strategy is flexible in the amount of work it does and the number of rules it generates, but because of its more thorough examination of at least part of the data it can choose to retain a more appropriate subset of the rulebase. It can also optimise range conditions in rules to maximise antecedents and minimise consequents by taking account of *empty subranges* (query ranges include an arbitrary amount of empty subrange which gets copied into rules by earlier query triggering systems).

New topics in this paper include the use of systematic data analysis, guided by criteria relevant to the query-processing application and information discovered during analysis.; and the Condition Dependency Graph model, which provides a uniform framework for query processing, rule production and rule management.

References

1. C. Hsu, and C.A. Knoblock, Rule Induction for Semantic Query Optimization, Proc 11th Intl Conf on Machine Learning, 1994, pp112-120.
2. C. Hsu, and C.A. Knoblock, Estimating the Robustness of Discovered Knowledge, Proc 1st Intl Conf on Knowledge Discovery and Data Mining, 1995.
3. B. G. T. Lowden, J. Robinson, and K. Y. Lim, A Semantic Query Optimiser using Automatic Rule Derivation, Proc WITS 95, 5th Intl Wkshop on Information Technologies and Systems, Holland, Dec 95, pp 68-76.
4. A Sayli, and B G T Lowden, The Use of Statistics in Semantic Query Optimisation, Proc. 13th. European Meeting on Cybernetics and Systems Research, pp 991-996, Vienna, 1996.
5. S. Shekhar, B. Hamidzadeh, A. Kohli, M. Coyle, Learning Transformation Rules for Semantic Query Optimization: A Data-Driven Approach, IEEE Trans Data and Knowledge Engineering 5(6) 950-964, 1993.
6. 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.
7. M. Siegel, E. Sciore, S. Salveter, A Method for Automatic Rule Derivation to Support Semantic Query Optimization, ACM TODS 17(4) 563-600, 1992.