Semantic Inequivalence:
A Link Between Knowledge Discovery and Query Optimisation

By Meera Khurana

Birkbeck College

University of London

For the Degree of

Master of Philosophy

Declaration

I declare that the work presented in this thesis is my own.

Abstract

This thesis investigates linking the output produced from data mining of association rules from a database and the database management system's query optimiser, with the aim of supporting improved query optimisation. A novel query optimisation technique, Semantic Inequivalence (SI), is presented based on this idea.

The thesis critically reviews data mining of association rules, query processing and optimisation, with emphasis on research into the areas that are most relevant to the new idea that the thesis introduces.

SI is described and how it differs from other query processing techniques is explained. This new concept is also exemplified using sample queries to aid understanding, and it is shown how the output from the data mining of association rules can be used in conjunction with query processing in a complementary way.

An extensible optimiser is reviewed in terms of its structure and how SI can be incorporated into it. The thesis goes on to formally define SI within the context of an extensible query optimiser. Its input, processing and output are formally defined.

An algorithm that implements SI is presented. The costs and savings of using it within query processing are compared to the costs of not using it. Exceptional and rare cases are also investigated from an empirical point of view. SI is also discussed in detail within the context of the established *Decomposition Algorithm* for query processing.

Empirical evidence of the usefulness of SI is analysed, by using the presented algorithm for actual queries posed to real-world large databases with related association rules. As well as empirical analysis with real data, SI is studied with

synthetic data based on the normal distribution. In all situations the I/O costs of using SI is compared to not using SI. Thus the most beneficial scenarios for using SI are documented, as well as those situations that do not reap any advantage.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction and Thesis Overview

1.1 A Dynamic and Collaborative Computing Environment

This thesis brings together two areas related to database management systems that have so far been kept relatively distinct from each other. These are the areas of query optimisation and data mining of association rules.

Techniques for query optimisation have been developed which exploit semantic information about a database derived from integrity constraints [27]. The scope for query optimisation to exploit association rules (or simply *rules*) has not been pursued so fully. This thesis focuses on the exploitation of such rules in query optimisation.

This thesis demonstrates the benefits that can be achieved when rules are used to enhance query optimisation. The rules can be exploited by the database's query optimiser because they can contribute to reducing the response time, based on comparing I/O, of queries that would otherwise take significantly longer to execute. This is demonstrated in the thesis by using real-world databases and queries, in addition to using synthetic data and queries. An independent costing algorithm, based on breaking down a query, rather than I/O required, is also used to show the benefits of the new approach.

1.2 Motivation and Aims

The idea for this thesis originated from the fact that increasingly organisations are implementing data warehouses using relational database management technology [8]. Hence there is increasing demand for finding useful knowledge from data warehouses that can provide otherwise unknown information about an organisation, a particular industry or customer preferences. This can enhance the competitive advantage of a company and influence decision making; this explains the significant interest shown from industry in this area [8].

The motivation for bringing together the two aforesaid areas came from studying the vast amount of research on efficient rule discovery techniques, and the potential of how the resultant output can be used by a database management system. This thesis views query optimisation as an area that can benefit from the association rules that are output in order to improve query response time. This has advantages for the user, the organisation and for current and previous research in terms of faster response times, discovery of information and using the results of data mining of association rules in novel ways for furthering research, respectively.

1.3 Example of Using an Association Rule with a Database Query

If from the available association rules, we know that given the value of column *A*, we can determine the value of column *B*, or:

```
if A = value_a then B = value_b (80% confidence)
```

If a query is:

```
SELECT DISTINCT B
FROM  table1
```

```
WHERE  A = 'value_a'
```

This can be re-written to take this defined relationship into account, and only retrieve what is now unknown, by taking the information that the associated rule provides into account.

The rule answers 80% of the query. This is because the query optimiser can know from the rule that 80% of the values of the requested column, *B,* which is being retrieved has the value of *value_b*. Therefore it can eliminate this pre-defined partial result from the query that is executed by modifying the query to only select the rows where column *B* does not have the value *value_b.*

The query below demonstrates how the concepts introduced in the thesis propose taking this known information from the association rule into account and using it in the query optimisation strategy to partly answer the query. It does not execute the whole query against the database, but only the subset or part that we do not know from the defined association rule. The resulting query to process is:

```
SELECT DISTINCT B
FROM  table1
WHERE  A = 'value_a'
AND    B <> 'value_b'
```

The resultant query takes the known information about the pertinent data relationship into account, meaning that it only asks for information that is both originally requested *and* not otherwise known.

1.4 Contributions

The thesis introduces a new query processing strategy, ***Semantic Inequivalence*** (SI), that intentionally uses inequivalent queries as part of the transformation process rather than previously researched query processing extensions [1, 24,

41], which emphasise the importance of semantically *equivalent* query transformations [36].

SI can use association rules with 100% or less than 100% confidence to help answer queries in a number of different ways. Where association rules have less that 100% confidence, the thesis looks at how they can be used in conjunction with existing research to build upon and further query optimisation.

An algorithm that implements SI is presented. The algorithm is defined for use with select-project queries excluding aggregates, group by and having clauses. Empirical evidence of the usefulness of SI is established with analysis of the results of using the algorithm with two real data sets and one synthetic data set.

1.5 Structure of the Thesis

The structure and layout of the thesis is now detailed.

Chapter 2 provides an overview of data mining for association rules, and discusses the scope for its use in query optimisation. Query processing and optimisation are outlined, as are various aspects of query processing research in areas that are most relevant to SI.

Chapter 3 defines the new concept ‘Semantic Inequivalence’. It goes on to discuss how SI differs from other techniques. A data model is then defined upon which the example queries given throughout the thesis are based. Examples within the chapter demonstrate how the output from data mining association rules can be exploited in query processing. The use of SI with rules with 100% or less than 100% confidence is discussed.

Chapter 4 presents a detailed specification of the SI algorithm. The input, processing and output are formally defined. The resulting optimiser is hereafter referred to as the *SI algorithm*.

Chapter 5 looks at the cost of using SI because for SI to be a useful query processing technique, it should be able to reduce the cost of answering queries in some definable situations. The costs of using SI are compared to the costs of not using it. Exceptional and rare cases are also looked at. For the purpose of demonstrating the effects of applying the SI algorithm on query processing cost, the cost of answering queries in their original form is compared with the cost of using the corresponding SI queries. This is analysed in order to identify the situations where SI adds the greatest advantage.

Chapter 6 focuses on empirical evidence of the usefulness of SI, by looking at using the SI algorithm for real-world queries with two real large databases. Association rules are found from the databases that are relevant to the queries executed. This provides details on the usefulness of SI by posing actual queries against large independent databases. The reasons for choosing two distinct real-world databases are given and the cost outputs that the query optimiser produces for the original and SI transformed queries are compared.

Chapter 7 focuses on synthetic data distribution – based on the normal distribution - and uses the SI algorithm. Thus this algorithm is also evaluated in a controlled environment. It demonstrates how various locations of the data mining association rule antecedent on the distribution can affect the use of the said algorithm in terms of query costing for original and transformed queries.

Chapter 8, the concluding chapter, presents a summary of what the thesis has discussed and achieved. Additionally, it identifies potential areas for further related research.

Appendix A1 defines terms and keywords that are used within the thesis.

Appendix A2 defines the different typefaces that are used within the thesis.

Appendix A3 defines the abbreviations that are used within the thesis.

Appendix A4, lists all the output that the SI algorithm produces for each of the queries used to exemplify SI in Chapters 6 and 7. It lists the optimiser's chosen query plans and I/O.

Appendix A5 discusses SI within the context of an independent query costing algorithm - the established Decomposition Algorithm for query processing. This autonomously demonstrates how SI can reduce the query processing cost in a generic way, without using any specific query example.

Chapter 2

Data Mining of Association Rules and Query Optimisation: Background

2.1 Introduction

This chapter provides an overview of the data mining of association rules and looks at the main focus of previous research in this area. It discusses the current research limitations and how the boundaries can be extended to improve the usage of the potentially very valuable information that data mining of association rules can produce for use in query optimisation. A clear distinction is made between association rules, functional dependencies and integrity constraints. This is followed by a discussion of the query optimisation process and existing work on semantic query optimisation. Finally an approach for the exploitation of data mining rules within an extensible query optimiser is introduced.

2.2 Data Mining of Association Rules

A large volume of knowledge that may exist can be discovered from a database [20]. There are ‘intelligent’ ways to associate a query with the discoverable database knowledge [20]. Interest from industry in this area is due to the fact that data mining may result in knowledge that can be of vital importance for a company [8]. Some useful semantic data patterns that exist in a database are simply not known [16]. If they were known they may have been implemented as

database integrity constraints. This is however unnecessary if they do not need to hold true, but simply reflect the current state of the data held in the database.

Data Mining of Association Rules can be defined as a process for discovering association rules from a large database – it is also known as ***knowledge discovery*** [28].

(Unless otherwise stated, the term data mining when used throughout the thesis refers to data mining of association rules).

2.3 Rules and Functional Dependencies

A rule is not necessarily the same as a *functional dependency*.

A ***functional dependency*** states that the value of an attribute (or set of attributes) is uniquely determined by the value of some other attribute (or set of attributes) [20]. Hence, given the value of an attribute *A*, then the value of attribute *B* can be determined with 100% confidence. If a rule has 100% confidence it may be called a *functional dependency*. Otherwise it is an *approximate dependency* [20].

An ***approximate dependency*** is a functional dependency that almost holds. Some rows can contain exceptions to the stated dependency. This is an alternative name for an association rule [30].

Therefore association rules provide information on data patterns within the database, *plus* the probability of them occurring.

Integrity constraints enforce the data values that are acceptable for certain attributes. In this sense they are like a pre-defined rule. In contrast, association rules do not protect the integrity of the data, or enforce particular data values but rather characterise the current database environment [16].

An ***association rule*** (or simply a rule) is of the form X → Y, where X = {x_1, x_2, …x_n} and Y = {y_1, y_2,…y_m} are sets of items with x_i and y_j being distinct items for all *i* where $1 \leq i \leq n$ and all *j* where $1 \leq j \leq m$. The database is considered as a set of transactions, each transaction containing such items. Any association rule has the form LHS (left-hand side) → RHS (right-hand side), where LHS and RHS are sets of items. The LHS is the ***antecedent*** of the rule. The RHS is the ***consequent*** of the rule. The set (LHS ∪ RHS) is called an ***itemset***. The meaning of a rule is that a transaction that contains the antecedent set of items also contains the consequent set.

In the context of a relational table, X and Y may correspond to values of columns within the table. For example, a rule might be:

`If` *`column_X = value_X`* `then` *`column_Y = value_Y`*

This means that a row which has *value_X* for *column_X* also has *value_Y* for *column_Y*.

For a rule to be of interest it should meet some interest criteria. Two measures for this are ***support*** and ***confidence.*** Each itemset has these two associated statistics [2, 30].

A rule's ***support*** is the proportion of rows that contain both the rule's antecedent and the rule's consequent [12]:
(LHS ∪ RHS) / total number of rows

A rule's ***confidence*** is defined as the number of rows that contain both the rule's antecedent and the rule's consequent divided by the number of rows that contain the rule's antecedent [12]:
(LHS ∪ RHS) / number of rows with LHS

This is the probability that a row contains both an antecedent and consequent of the rule, given that the antecedent occurs. The confidence statistic is the measure of a rule's strength.

The confidence and support statistics of a rule are important in the data mining process because they are used as a way to filter rules so that only the most prominent ones - those which have a minimum user-specified level of support and confidence - are output [36]. This is in order to prevent the output of data mining being unnecessarily or unmanageably large [36], given that output can be vast. Therefore, data mining rules used by the query optimiser should be limited to those that are most useful to the database usage pattern.

Association rules are used throughout the thesis with the format:
exp_X → *exp_Y* (N% confidence)

Both *exp_X* and *exp_Y* are *expressions*.

An *expression* has the form:
<operand><operator><value>

Where:
<operand> is a column
<operator> is in { '=', '<', '>', '<>', '>=', '<=' }
<value> is a literal value

Columns that exist in the rule's antecedent cannot also exist in the rule's consequent and vice versa.

2.4 Data Mining Research and Scope for Use in Query Optimisation

The data mining of association rules can be a very expensive task in terms of I/O, processing power and time required [2, 30, 38]. Therefore, data mining research has very much focused on the development of algorithms that can find rules that exist in a very large DBMS as quickly and efficiently as possible. If a process is too expensive it reduces its profitability and worthiness - the advantage of it compared to its cost – thereby reducing its usefulness. This has given rise to some prominent data mining algorithms, including Apriori [2, 42], AprioriTid [2], Partition algorithms [30] and Hash based algorithms [34].

The aim of these algorithms is to find as efficiently as possible, predominantly by minimising I/O cost, associations that exist between data items in a database that meet the minimum support and confidence criteria. This can be approached by breaking the task down into two problems [2]. The first problem is to find all sets of items that have support above the user-declared minimum support. These are referred to as *large itemsets*. This is subsequently used as the input to the second part of the problem, which is to find all of the rules meeting the minimum confidence level. This is solved by taking each large itemset generated in solving the first problem, and for each large itemset, finding 2 subsets within the large itemset, such that there is no overlap in the attributes in each.

Hence rules are derived from the large itemsets. Algorithms that are based on this two-phased approach include Apriori, AprioriTid and Partition, plus hybrids or variants of these. Regardless of the algorithm used, the output is a set of rules meeting the minimum support and minimum confidence criteria.

The rules discovered could be actual business rules that are unknown, and hence have not been explicitly specified. Alternatively, rather than unspecified business rules, they may be what ‘just so happens’ to be the case in the environment represented by the database.

Even though data mining can discover potentially useful data attribute relationships, hence information, and even increase knowledge of the database subject's environment, research has so far very much focused on algorithms to produce the output rather than on how to use the output. This thesis proposes using association rules to optimise user queries.

The discovery of unexpected association rules would be a useful deployment of data mining applications [20]. This makes the need for automatic usage of the output more important. An area of the database management system, which could benefit from the otherwise unknown, or newly acquired knowledge of data patterns or data relationships among attributes, is the query optimiser.

2.5 Query Processing and Optimisation

Query optimisation is the process of analysing a query, finding out what resources are required to answer it and how the resources can be reduced to answer the query more efficiently [45]. Query optimisation is often performed in two phases: a logical optimisation phase and a physical optimisation phase [8].

During logical query optimisation, the order in which query operations are performed is determined. The physical query optimisation then determines how the operations can be most efficiently performed [20]. This depends on the way the data is stored – its physical schema. Physical optimisation is performed with respect to a cost model. This involves searching alternative access paths for accessing the database objects. However, when the search space is very large, considering all possible alternatives may not be feasible due to the time and resources required. Hence a control strategy based on a heuristic approach may be used [14].

Modern optimisers are cost based, rather than syntax based. Input to the optimiser includes the parsed query, often in Structured Query Language (SQL) [32, 45], along with information on the database objects. This may include the size of each table in the query, the indexes on the tables, if any, and the type of each index, the columns used in the query, the density and distribution of the indexed columns, join ordering, the use of internal temporary tables, available data cache and the physical I/O sizes supported [45]. The list depends on the sophistication of the optimiser's costing algorithms.

The methods of executing a query are determined and the cost of each found. The cheapest, most efficient method is selected by the optimiser to execute the query.

2.6 Semantic Query Optimisation

The area of semantic query optimisation (SQO) has been well researched [1, 27, 39, 40, 51]. This is where a query is transformed based on functional dependencies known about the data [1]. SQO maintains the semantics or meaning of the query – therefore it produces semantic equivalent queries only [41]. A transformed query is equivalent to the original one if it gives the same answer for every legal database state [4].

SQO is achieved by adding a constraint to a query based on a rule with 100% confidence. The consequent of the rule is added as an additional predicate to the query [41]. Hence SQO can only be used with functional dependencies and cannot make use of approximate dependencies. Since the rules that SQO can use must have 100% confidence, its usability is limited because of not being able to use high confidence rules, such as those with 80% or 90% confidence levels. In fact, SQO relies upon data constraints rather than the data values reflected in the database state. Even association rules discovered with 100% confidence are not sufficient for SQO – because they may not always be satisfied unless the data values are explicitly constrained.

Trigoni and Moody [47, 48] take a different approach to SQO, in that they look at using association rules rather than functional dependencies. However they do not discuss the probability of rules. This is important when transforming a query since if the probability is not 100% the semantics will change. They therefore assume the association rules always hold, hence effectively treating them as functional dependencies. Additionally, [48] removes predicates from a query if they are implied by other existing predicates. However, removing predicates reduces the information supplied to the optimiser. This can reduce the query paths that the optimiser considers, which may increase the cost of answering a query.

2.7 Data Mining Rules

Data mining produces information regarding the relationships among data items stored in a database [2]. Data mining can produce a large quantity of output [38]. Therefore, its output needs to be filtered to those rules that are most useful to the applications or query profiles against the database. This requires local knowledge of data usage as an input to the rule filtering process.

While the query optimiser tries to find the cheapest way to access data [32], it does not currently have the option or availability of using of a ***rule page*** to help process a query. The concept of a rule page is introduced in this thesis. It is a page managed by the DBMS, which holds the most useful rules relevant to a database's querying patterns.

Just as database management systems have data pages, index pages, statistics pages, etc. [45], rule pages are similarly used specifically for storing one type of data: rules.

An example of a rule is:

```
if age = 30 then residence = 'UK' (70% confidence)
```

This would be stored on a rule page rather than on part of the database's data, index or statistics pages. Section 3.9 provides an example of the contents of a rule page.

Rules to be held on the rule page can be determined by a database administrator with local knowledge of data usage. Filtering the rules is important to avoid irrelevant rules from consuming space in the rule page. Alternatively, *templates* can be used to describe the pattern of a useful rule [36], so that only those that meet the template pattern are considered useful, or 'interesting' with respect to the environment. Either way, the aim is to prevent too many rules, or 'rule overload', some of which may not be useful to the environment. Having too many 'non-useful' rules would compromise the usefulness of the concept as more pages would be required for storing the 'non-useful' rules. This would subsequently result in more I/O required to access the useful rules. The information in a rule page may answer a query, or the query optimiser may have a module for re-writing the query into a subset of the original query, taking into account what is known from the rules. This can be used to answer a subset of the original query. This results in a 'narrower' query to be answered by the DBMS's query processor.

2.8 Extending the Query Optimiser

For SI to be a new strategy for the optimisation process, it needs to be incorporated into the query optimiser. To enable this, the DBMS's optimiser requires the flexibility of adding new optimisation strategies or techniques.

Optimiser extensibility refers to the adding of, or the ability to add, new query processing strategies to the database management system's optimiser [10, 31].

The *strategies* are also known as query optimiser *components* [31]. This thesis goes one step further from recent research in new query processing strategies because it uses inequivalent queries as part of the transformation process rather than previously researched extensions [1, 28, 46], which emphasise the importance of semantically equivalent query transformations [14, 41].

SI is a new strategy in query optimisation. To study SI in more detail an extensible optimiser based approach will be used. This means that the optimiser has a structure that incorporates the flexibility for adding new strategies in optimisation. This is achieved by its having a modular or component-based architecture, where each *module* (also called a *region*) has a particular goal or strategy in the query optimisation process.

Regions are effectively query transformations [31]. They take a query as input, transform it using the region's particular query transformation strategy, and produce the transformed query as the region's output.

Because each region has a particular strategy or goal, for a query optimisation region this would mean transforming the input query into an output query that has lower execution cost.

In an extensible optimiser, a new query optimisation strategy, such as SI, can be added to the optimiser by adding a new region or module to the existing structure. This region then needs to be integrated into the database management system's optimiser. This may be achieved by using a hierarchical control structure between regions. Hence, the optimiser's global control region (the parent region) sends the query being processed to the child regions for transformations until a final form of the query is produced for execution. The hierarchical structure of an extensible optimiser is represented in Figure 2.1.

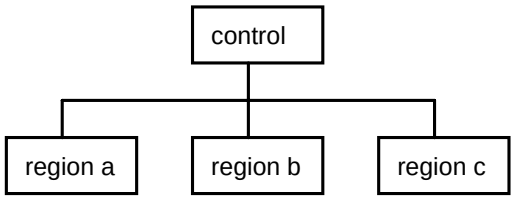

Figure 2.1 – Extensible Optimiser Structure

Using the modular, extensible approach, each region embodies a strategy for achieving a particular goal in the query optimisation process. For example, there may be a region to find the most efficient way of ordering the joining of tables in a query. Another region may have a goal to find the best access strategy for each table.

There is a *parental* or *control* region which is responsible for deciding which region/s to send the query to for transformation and hence manipulation.
Therefore, SI processing can be incorporated by adding a region to an extensible optimiser. The region can transform the query into an SI query, if considered appropriate by the parent region.

The controlling, parent region decides which subordinate regions should be used to transform a query. To enable this, each region needs to be defined unambiguously for the parent to decide whether it is suitable to be used for a query.

To facilitate this, the definition of a region should consist of the following:
1. A description of the set of possible input queries that the region can accept for transformations.
2. The set of transformations that can be performed on the queries.
3. A goal that characterises the output query produced by the region. This is what the region aims to achieve.

The concept of *regionalised* or *modular* optimisers provides a great deal of flexibility because it enables new query optimisation processes to be added by adding a region with an interface that the optimiser understands. This type of

structure is ideal for adding SI to.

The architecture of a region, illustrated in Figure 2.2, has two parts:

1. A *control* (or *implementation*) and
2. An *interface* to the parent region in the hierarchy.

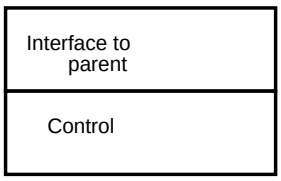

Figure 2.2 – Region Architecture

The interface to the parent region is needed in order to communicate what type of queries it can process. This is required to determine which types of queries can be passed as input, and what the region's goal or purpose of its existence is.

The interface informs the parent control about what the region can do and whether it is suitable for the query. The parent uses this information to decide which regions the input query should be passed through for manipulation.

The implementation of the region is the region's *control* section. This is the encapsulation of the region's achieving its goal. It is the embodiment of the region's strategy and manipulation of the queries passed into it. The control does the decision taking and transforming of the query. Hence, via the interface to the parent, the region receives a query to transform. The region's control will then convert the query, as implementation dictates.

An extensible optimiser allows adding a region with the only requirement that the interface can communicate with the parent. Therefore, adding a region causes minimal disruption to the existing optimiser features embodied as other regions.

In summary, a region is essentially a query transformation. It has a control

strategy for applying transformations and produces a transformed query as output.

2.9 Conclusion

This chapter has given a brief overview of data mining of association rules and outlined the main focus of research in this area. It has highlighted how the data mining output can be a useful input to the database query optimisation process. This is by taking the view that there are intelligent ways to associate a query with the discoverable database knowledge [20] that are represented as association rules.

For SI to be used as a component of query processing, there should be known patterns or association rules between data items present in the database.

The information about data relationships may be viewed as a ‘gap’ in the input to the query optimiser that can be filled by data mining output. It takes the view that this can add value to the query optimisation procedure. By connecting these two separately well-researched areas together we produce a query optimisation opportunity believed to be highly valuable, as will be seen throughout the thesis.

The chapter distinguished semantic query optimisation, and its limitation regarding usability. The concept of a rule page was also introduced and how query optimiser extensibility is required for adding new query processing techniques.

Chapter 3

Semantic Inequivalence

3.1 Introduction

Chapter 3 defines SI, discusses its concepts, and demonstrates how SI can be used with practical examples. A data model is set up which subsequent example queries throughout the thesis are based upon.

The chapter goes on to discuss how the association rules that may be produced from data mining, or any other source, can be used as an input to the SI process and the potential gains that can be achieved by using the rules in this way. The aim of SI bringing together data mining of association rules and query optimisation is to achieve increased query selectivity resulting in more efficient query processing possibilities. Instead of pursuing the overhead of repeatedly requesting known information, SI endeavours to make use of the rules or patterns known and defined about the data values, with the aim of reducing the cost of answering the original query.

Following this, the chapter goes on to look at some of the different ways in which SI can be used in conjunction with other query processing techniques, such as partial indexing [37].

Lastly, changes in the confidence levels of association rules are reviewed, followed by the chapter's concluding remarks.

3.2 Definition of Semantic Inequivalence

Definition:

Semantic Inequivalence (SI) is a process of transforming a query, *Q*, into another query, *Q'*, by taking known information about the relationships between column values that are stored in the database into account.

Let Q_r denote the result rows of query *Q* on relation *r*. The same obtains for Q_r'.

Assume association rule, *S*, implies the existence of rows Q_{rs} in the result rows Q_r.

SI transforms *Q* to *Q'* such that $Q_r = Q_r' \cup Q_{rs}$.

Explanation:

SI intentionally changes the meaning, or the semantics of the query *Q*, to only request the unknown part of the query, *Q'*, the rest being known and hence can be answered from the database's defined association rules.

The term *Semantic Inequivalence* is based on the idea of changing the meaning of a database query, *Q*, to a subset of *Q*, represented by *Q'*, which only inquires for that part of the original query that is unknown from information (the *association rules*) held about the column data values in the database. Therefore the result of the query that is executed, *Q'*, is a subset of the result of the original query, *Q*. The rows that are not requested by *Q'* but are requested by *Q* is the implicit information that is provided by the defined association rule, and hence can be answered without additional data retrieval. Consequently, the semantics, or the meaning of the query that is executed is different to that of the originally requested query.

3.3 How Semantic Inequivalence Differs from Syntactic Query Optimisation

Syntactic query optimisation is based on transforming a query into another one, which has the same result set as the original query but can be processed more efficiently [25, 46]. The syntactical query optimisation algorithm takes as input a query, *Q*, submitted by a user and aborts it if a contradiction is found, or otherwise returns as output an equivalent query *Q''*, which produces the same answer as *Q*. Syntactical query optimisation can only use 100% confidence rules and not approximate dependencies.

This thesis goes one step further from previous research in new query processing strategies, because it intentionally uses inequivalent queries as part of the transformation process rather than previously researched query processing extensions [1, 28, 46], which emphasise the importance of semantically *equivalent* query transformations [14, 41].

SI can use association rules with 100% or less than 100% confidence to help answer queries in a number of different ways. This is seen in the following section.

3.4 Using Semantic Inequivalance With Association Rules

This section looks at some ways in which SI can be used in conjunction with database association rules.

Firstly, if a rule has 100% confidence it may be used to completely answer a query. This may be referred to as ***rule covering***. For rule covered queries, neither

the data pages nor the index pages need to be accessed. For this to be the case, both, the *select* list of the query and the column predicate in the *where* clause (assuming queries written in SQL) need to be in the rule as the consequent and antecedent, respectively. This is discussed further in Section 3.10.

Similarly, yet conversely, a 100% confidence rule may also be used to return a null result set very quickly. This would be the case where a contradiction is found between the query and the rule, even where a rule does not cover the query. This is known as ***incoherence detection*** [4], and can be a powerful application of SI in terms of reducing the query processing costs.

Secondly, a rule, with either 100% or less than 100% confidence, can be used to partially answer a query (where in the former case the rule does not cover the query). The query can be re-written into an SI one that answers only the subset of the original query that the rule does not answer. If a query has a very large partial result set such that it takes a long time to execute, the result from the rule can be returned first. This provides the user at least with a partial result to look at immediately.

In yet a third way, rules can be used in conjunction with partial indexing [37]. Rather than index a whole table, ***partial indexes*** only index a portion of rows in a table. That is, they only index those values that are not part of a large repeating group. Repeating data values in columns is the sort of skew in the data that an association rule would define. Hence the rules can replace the indexing of non-indexed covered values. This helps performance [37] by reducing I/O and reducing the object's storage requirements by having smaller indexes since the frequent or commonly occurring values are not repeatedly indexed. Otherwise these would be included many times in the index – once per record occurrence. This is complementary to SI because partial indexes have fairly extreme selectivity involving non-repetitive values, and conversely association rules with a significant level of support involve repetitive values. The use of SI with partial indexing is complementary in that the query optimiser can use rules for the highly repetitive values, and partial indexing for the less repetitive values. The less repetitive values are those that would not be covered by a rule. Both

techniques, SI and partial indexing, are based on a non-uniform data distribution, or in other words a skewed data distribution.

By SI being able to use rules with less than 100% confidence, a major restriction on situations where rules can be used is removed, or at least, reduced.

3.4.1 Semantic Inequivalence Example

This section looks at a sample query based on a set of tables reflecting a simple database structure. SQL is used with select-project queries, excluding aggregates, group by, having and order by clauses.

Database Structure:

```
table1    (A, B, C, D)
table2    (A, E, F, G)
table3    (E, H, I, J)
```

The letters *A* to *J* represent columns. The underlined columns are primary keys. The columns *A* and *E* are foreign keys in tables *table2* and *table3*, respectively.

Sample Query:

This is based on the simple relational database structure, and selecting some columns across the tables requiring joins between them. The query optimiser has no information other than what is provided by the query and the known database structure.

A sample query to pick out information from the tables would be:

```
SELECT t1.B, t3.I
FROM table1 t1, table2 t2, table3 t3
WHERE t1.A = t2.A
AND  t2.E = t3.E
```

```
AND  t1.A = value_a
```

Sample Query Explanation:

Looking at this SQL query, the variables defined are *t1, t2* and *t3*. These represent the ranges *table1, table2* and *table3,* respectively. The qualification or *where* clause is a Boolean function *fn(table1 * table2 * table3)* which is the product of *table1, table2* and *table3*. The output of the function *fn(table1 * table2 * table3)* is applied to the function *fn(A = value_a) = true*. The columns retrieved from the resulting table are *B* and *I*.

The proposal is for data mining output to be used to extend and hence modify the sample query to take known column values into account as detailed in example 3.2. This may open up multiple methods for processing the query, with different data access paths, having the goal of reducing the cost of the query's execution.

3.5 Demonstrating Semantic Inequivalence

If from the available association rules, we know that given the value of column *A*, we can determine the value of column *B*, or:

```
if A = value_a then B = value_b (80% confidence)
```

If a query, *Q* is:

```
SELECT DISTINCT B
FROM  table1
WHERE  A = value_a
```

This can be re-written to take this defined relationship into account, and only retrieve what is now unknown, by taking the information that the associated rule

provides into account.

To form the complete query result, the result set of the SI can be formed with a union of the result set that is known from the rules.

The result set that is known from the rules is what is saved from being processed unnecessarily through the query optimisation and data access and retrieval processes.

The association rules give extra and maybe even new information on the relationships between database attributes. For example, to say that column *B* in *table1* depends on the value of column *A* in *table1* is the same as saying that column *A* determines the value of column *B*.

Example 3.1:

Based on the above query, with the stated rule:

```
if A = value_a then B = value_b (80% confidence)
```

Let the following sample data values exist in table *table1*.

A	B	C	D
value_a	value_b	value_c	value_d
value_a	value_b	value_c2	value_d
value_a	value_b	value_c3	value_d
value_a	value_b	value_c3	value_d
value_a	value_b	value_c2	value_d
value_a	value_b	value_c1	value_d
value_a	value_b	value_c1	value_d
value_a	value_b	value_c1	value_d1
value_a	value_b1	value_c2	value_d1
value_a	value_b2	value_c	value_d2

Table 3.1 – Sample Data Values

Using the query, *Q*:

```
SELECT DISTINCT B
FROM  table1
WHERE  A = value_a
```

The rule answers 80% of the query. This is because the query optimiser can know from the rule that 80% of the requested column, *B*, which is being retrieved has the value of *value_b*. Therefore it can eliminate this pre-defined partial result from the query that is executed by modifying the query to only select the rows where column *B* does not have the value *value_b*.

The query below demonstrates how SI proposes taking this known information from the association rule into account and using it in the query optimisation strategy to partly answer the query. It does not execute the whole query against the database, but only the subset or part that we do not know from the defined association rule. The reduction in data that is being requested can lead to increased efficiency if less I/O is required to answer the query. This is discussed in detail in Chapter 4. The resulting SI query is:

```
SELECT DISTINCT B
FROM  table1
WHERE  A = value_a
AND    B <> value_b
```

This is a simple style example that provides a practical overview of the concept of SI.

Example 3.2:

Based on the query defined in Section 3.6.1,

Original query:

```
SELECT t1.B, t3.I
FROM table1 t1, table2 t2, table3 t3
WHERE t1.A = t2.A
AND  t2.E = t3.E
AND  t1.A = value_a
```

Applying SI to the original query, by taking what is known into account from the example association rule:

```
if A = value_a then D = value_d (70% confidence)
```

then we have the following SI query:

SI Query:

```
SELECT t1.B, t3.I
FROM table1 t1, table2 t2, table3 t3
WHERE t1.A = t2.A
AND  t2.E = t3.E
AND  t1.A = value_a
AND  t1.d = value_d
UNION
SELECT t1.B, t3.I
FROM table1 t1, table2 t2, table3 t3
WHERE t1.A = t2.A
AND  t2.E = t3.E
AND  t1.A = value_a
AND  t1.d <> value_d
```

This new query may be particularly useful in determining results in a very large database if there is an index on column *D*, or alternatively a composite index on columns *(A, D), (D, A) or (D, another_column).* This transformation will be

beneficial if selectivity of the query is increased by the possibility that more access paths are made available. If the column, *D*, or columns *A* and *D* are indexed, then selectivity should be increased by the query otherwise there would be little benefit to having the index in the first place.

In this situation, syntactical query optimisation cannot be used because the rule does not have 100% confidence. This effectively ‘wastes’ the opportunity to use a highly confident rule that could improve query response time.

3.6 Data Model Example

The following data model example will be used throughout the thesis to exemplify the SI process.

It is based on publishers of books and their associated relationships with authors, titles, stores and customers.

A rectangle is used to indicate a table object.

An arrow indicates a one-many relationship between the objects it connects.

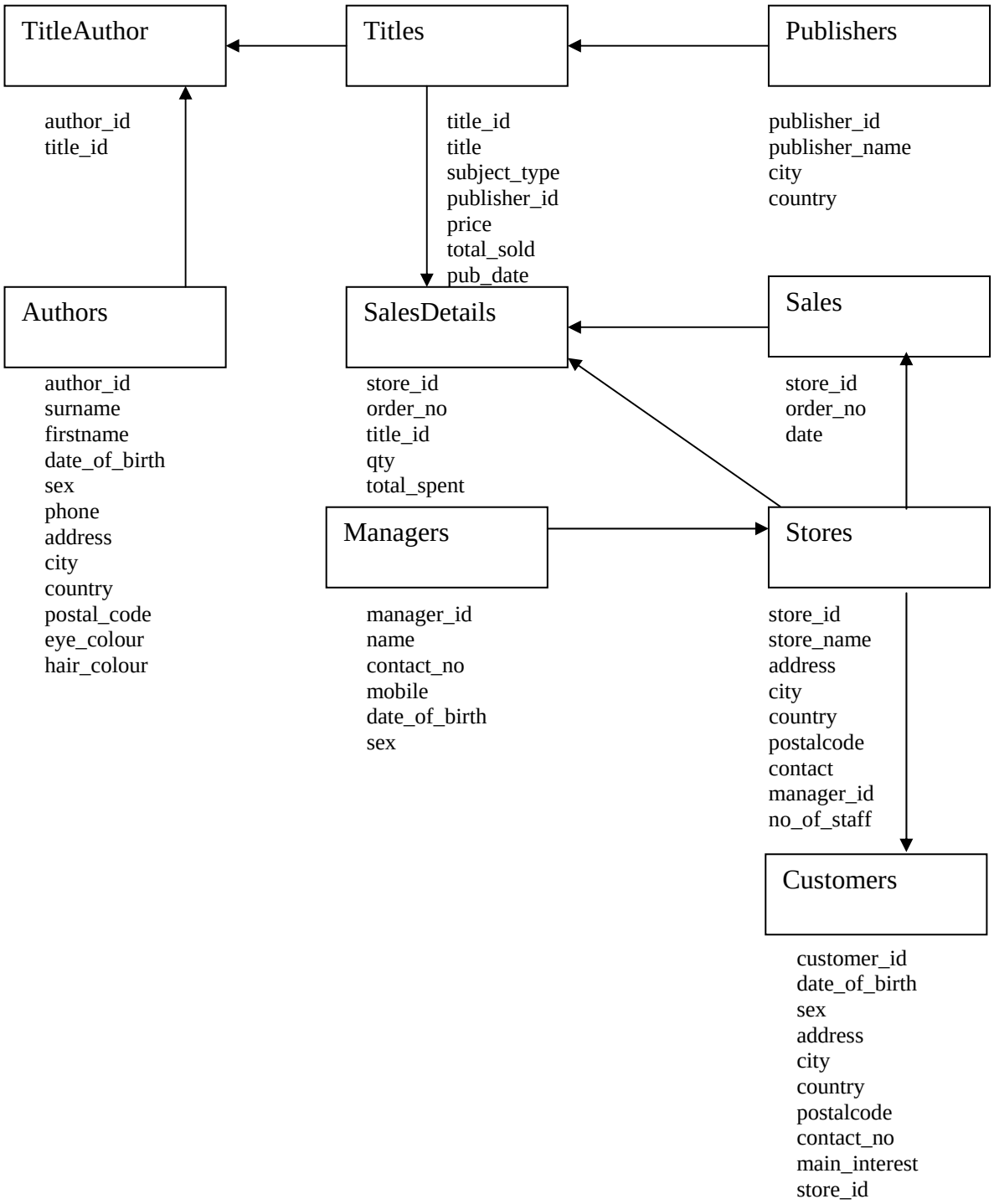

Figure 3.1 – Data Model

3.7 Data Mining Association Rules and Semantic Inequivalence Query Examples

Here are examples of association rules that may result from a data mining process executed on the above data model example instance that can be used as input to the SI process. They were derived manually by inspection. These rules serve as an example of the contents of a rule page.

```
if subject_type = 'cooking' then total_sold = 10000 (75% confidence)

if Stores.city = 'London' then Stores.country = 'UK' (100% confidence)

if subject_type = 'Astronomy' then price = 29.95 (70% confidence)

if title = 'Maths for beginners' then price = 15.00 (70% confidence)

if title = 'Maths for beginners' then total_sold = 100000 (75% confidence)

if qty = 5 then total_spent = 150 (80% confidence)

if eye_colour = 'brown' then hair_colour = 'brown' (70% confidence)

if Customers.city = 'NY' then main_interest = 'Financial' (65% confidence)

if Customers.city = 'Paris' then main_interest = 'Fashion' (70% confidence)
```

```
if store_name = 'The Bookshop' then no_of_staff = 6
(100% confidence)
```

As stated in Section 2.7, data mining can produce a large quantity of rules [38]. Hence filtering the rules to those that are most useful to the applications is important to avoid wasting rule page space that would slow down the optimisation process. For example, if it is known that all *city* attribute values are unique in the database so that the *country* attribute is not needed in order to determine the location, then the rules of the form:

```
if table.city = value then table.country = value
```

can be discarded.

Similarly, if there is no interest from the database users (no requirement for such information via business applications or user queries), in personal attributes of *authors*, then rules on their *hair_colour* and *eye_colour*, for example, may be discarded.

Example 3.3:

The following example also shows how the output of the data mining process can be useful for the SI process and additionally demonstrates how to reduce the cost of query execution.

Assume we want to know the prices of books based on the subject ‘Astronomy’. For this, we have a query, *Q*, with the following SQL:

Original Query:

```
SELECT DISTINCT price
FROM Titles
WHERE subject_type = 'Astronomy'
```

Knowing that 70% of Astronomy books have a price of 29.95 based on the rule:

```
if subject_type = 'Astronomy' then price = 29.95 (70% confidence)
```

we only need to ask for prices of Astronomy books where the price is not 29.95:

```
SELECT DISTINCT price
FROM Titles
WHERE subject_type = 'Astronomy'
AND price <> 29.95
```

This query requests 30% of the rows that have `subject_type = 'Astronomy'` rather than all of them since 70% are answered by the rule.

In the SI algorithm in Section 5.3, we will see that the actual predicate added would be:

```
(price < 29.95 or price > 29.95)
```

Therefore, if there are 500 books with `subject_type = 'Astronomy',` instead of accessing all 500 records, the SI query would access only 30% of the 500 records where `subject_type = 'Astronomy'`, which is 150 records compared to 500. This would require access to fewer rows than the original query. If there is an index on just the *price* column, especially if it is a clustered, B+ tree style index, or if there is a composite index on the *(subject, price)* columns, or on *(price, subject*), then applying SI would be a useful transformation, asking for only that part of the original query that is not answerable from rules. The new query is efficient if it has a lower execution cost, in terms of physical I/O, compared to the original query. The costing of the queries is detailed throughout Chapter 5.

3.8 Rules with 100% Confidence

In very large databases, data mining may produce some rules with 100% confidence. These may not be defined business rules with enforced integrity. They may be what 'just so happens' to be the case, or they may be 'unknown' business rules – rules that exist but are not actually known to the business or formally pre-defined.

In the case of rules with 100% confidence, these may be able to directly answer some queries, without requiring access to the database at all. This is an area that has been covered by existing research [41], and discussed in Section 2.6, where functional dependencies or 100% integrity constraint rules that are pre-defined can be used to help answer a query. This is done by adding the consequent part of the rule as an additional predicate to the query [1]. However, even though a 100% confidence rule may be used this way, it is not the same as a functional dependency or integrity constraint, because it may not be pre-defined or always be the case. It reflects the database state, which may change. Adding a new predicate may help the optimiser find a better access path by giving it more information. Moreover, if the column is indexed, a greater or improved choice of access paths may be available.

Work on semantic query optimisation encompasses the possibility that a query may be wholly answered from a rule, or more generally, an integrity constraint [27]. This thesis exploits this concept of 'rule covering' – where a rule 'covers' or completely answers a query, and does not require access to the data or index pages of the database. With 'rule covering', the rule is used to answer the query rather than to add a clause to potentially change the access path of a query to become efficient. The I/O associated with this is equal to the number of rule pages that need to be read. If there is a single rule page, then only 1 I/O is required. Hence, this can be an extremely powerful and efficient way of executing a query.

Example 3.4:

This example demonstrates the case of rule covering using a 100% confidence rule.

```
SELECT no_of_staff
FROM Stores
WHERE store_name = 'The Bookshop'
```

The rule:

```
if store_name = 'The Bookshop' then no_of_staff = 6
(100% confidence)
```

exists with 100% confidence, as defined, and is therefore suitable to answer the above query completely. This is an example of rule covering. It demonstrates the biggest possible advantage that rules can give when used together with the SI process.

The rule ‘covers’ or completely answers the query because both the predicate column and the column being selected are contained in the rule as the antecedent and consequent, respectively. With 100% confidence, there can be no other values contained in the query result set. That is:

$Q_{rs} \supseteq Q_r$

Example 3.5:

This example is also based on a 100% confidence rule, but does not cover the query. That is:

$Q_r \not\subset Q_{rs}$

Even if a rule with 100% confidence does not answer a query via rule covering, it may still be useful in the SI process. Let the table have an index on the *no_of_staff* column. Given the query:

```
SELECT  *
FROM Stores
WHERE store_name = 'The Bookshop'
```

From this query, we know that ‘The Bookshop’ is the only store name we are interested in from the *where* clause predicate. Using the same association rule as in example 3.4, on *store_name,* we know that *no_of_staff* = 6. Hence we can add this consequent clause so that the query becomes:

```
SELECT *
FROM Stores
WHERE store_name = 'The Bookshop'
AND no_of_staff = 6
```

In this situation, an existing index on *no_of_staff* can be used to access the table.

This strategy can also be used if there is a join in the query. For example, if there is a nested-loop join such that the new clause can be added to the outer table in the join, and the index is used to filter the rows in the outer table, then the inner table needs to be accessed fewer times. The next example query demonstrates this by searching for the names of the managers of stores called ‘The Bookshop’.

Example 3.6:

This example uses a join with a 100% rule that does not cover the query. This demonstrates that the SI principle can still be applied.

```
SELECT Managers.name
FROM Managers, Stores
WHERE Stores.manager_id = Managers.manager_id
AND store_name = 'The Bookshop'
```

As there is a predicate on the table, *Stores*, this can be used to filter rows from the table to join with the *Managers* table. If *Managers* is indexed on its primary key, *manager_id*, then this is likely to be the inner table of the join, so that for each row in the outer table the inner table can be accessed via the primary key index. If the query is as above, as there is no index on *stores.store_name*, the optimiser will need to fully scan the stores table. However, given the rule:

```
if  store_name = 'The Bookshop' then no_of_staff = 6
(100% confidence)
```

the optimiser can add the consequent as a predicate to the outer table, *Stores*. This way it has an index on the new column that can be used to access the table to filter the rows that it needs to use to join on the *manager_id* attribute of the inner table.

3.9 Physical Access Paths: Rules and Partial Indexing

The advantage of SI is most significant when a predicate is added that can change the data retrieval path to a more efficient one. For example, if the modified query results in the use of an appropriate index instead of a full table scan, this can reduce the required I/O. Detailed cost comparisons of different access paths resulting from changing the query are demonstrated in Section 5.4. Even if using the same access path for both the original and SI queries, the SI query may still be more efficient (for an indexed path) if the SI query accesses fewer index pages than the original query due to ‘reducing’ or narrowing the

query by requesting fewer rows, thus filtering rows that need to be accessed higher up the B-tree index.

3.10 Changes in the Confidence of Association Rules

The generation of association rules and their associated confidence is a relatively infrequently executed ‘batch’ style process rather than a frequent online process [36]. Since SI is oriented at very large databases, or data warehouses, the rules generated from data mining can take a significant amount of time to produce. Subsequently they should be filtered by the local database administrator’s knowledge regarding those rules that are useful to the environment. However, even though SI is meant for relatively static data and not for online transaction processing type of databases, the rule generation should not need to be performed often because the data is not intended for frequent change. Moreover, due to the large quantities of data held (100’s of gigabytes or terabytes) if a row needs an *ad hoc* change, it would not be considered significant enough to impact the rule’s probability. Hence association rules would not need re-calculating if there is no change to the rules’ confidence levels. This assumes that there are rules based on the columns that are updated. It is not considered cost-effective to maintain rules for each data change. There is the rare case whereby a 100% confidence rule no longer holds. However given this is a rare or extreme case, it is not explicitly dealt with. The change would be taken into account when the association rules are periodically refreshed.

If there is a requirement to make more frequent data changes such that the skew of the data does change, then a ‘subset’ process could be implemented to go through and verify the confidence of the association rules on the affected objects only. This would involve only re-calculating the rules on the affected objects - or even only the affected columns. The verification or re-calculation of rules on a per table or column basis would be significantly faster than full rule calculation database-wide.

Due to the above reasons, changes in association rule confidence are not dealt with in the thesis. If, however, despite the reasons above, it is considered necessary, it is a large enough area to be dealt with as another research topic in its own right.

3.11 Conclusion

This chapter has defined SI and given examples of how it can reduce a query to only request data that is not known from mined association rules held about column values in the database.

SI brings together database association rules, which may be output from data mining, with the query optimisation process, ensuring that generated rules are actually used to improve the efficiency of answering queries, by promoting automated usage of the rules.

The notion of SI promotes and is based upon the idea that association rules, held about the data in the database, can and should be used to help answer queries. This is reasonable given that the association rules are effectively data values that are already known to the DBMS. Hence when they can be used to form part of the query result set, the query optimiser should take advantage of this.

By adding more predicates to the query to eliminate requesting what is known, it has been demonstrated that the number of rows that need to be accessed by the SI query may be significantly less than the original query. This is useful if it opens up new data access paths, such as paths that enable the use of an index instead of a full table scan if this reduces the I/O required for answering a query. Section 3.10 demonstrates the special case of *rule covering*.

Rules with less than 100% confidence have also been shown to be very useful to the query optimiser. They can be used to partially answer a query and may be used in conjunction with other complementary processing techniques, such as partial indexing.

The data model example presented in this chapter is used for sample queries throughout the thesis.

Chapter 4

Semantic Inequivalence Algorithm

4.1 Introduction

While Chapter 3 gave an overview and examples of SI, this chapter provides a detailed specification of the SI algorithm. The algorithm is subsequently demonstrated using a sample query. This is followed by further illustrative examples of queries using the SI algorithm.

4.2 Algorithm of Control Section for Semantic Inequivalence Region

4.2.1 Notation used for Semantic Inequivalence Algorithm

Before defining the SI algorithm, this section introduces the notation used for the algorithm.

The query structure used is:

```
SELECT DISTINCT select_list
FROM   table
WHERE  predicate_list
```

Where:

`select_list` is the list or projection of columns requested by the query

`table` is the table used in the query

`predicate_list` is a conjunction of predicates $P_i \in \{P_1,\ldots,P_N\}$, in the query

A ***SARG,*** or ***search argument***, is a predicate in the form:

```
<operand><operator><value>
```

Where:

`<operand>` is a column

`<operator>` is in { ‘=’, ‘<’, ‘>’, ‘<>’, ‘≤’, ‘≥’ }

`<value>` is a literal value

Even though the input to the SI query is a conjunction of simple predicates, the output may be a disjunction of predicates.

The notation used:

Q represents the query

P_i and P_j represent predicates that are SARGs.

i, j represent variables with values *1* to *N*

C_i represents the column of P_i. Likewise, C_j represents the column of P_j

V_i represents the value of P_i. Likewise, V_j represents the value of P_j

P_i.*operator* is the operator used in P_i

C_i.*number_of_distinct_values* represents the number of unique values that the column C_i contains

C_i.*distinct_values* represents the set of unique values that the column C_i contains

An association rule, R, has the form:

```
If <ant> then <cons>
```

Where:

`ant` is the antecedent of the rule

`cons` is the consequent of the rule

`Ant` and `cons` have the form:

```
<operand> = <value>
```

Where:

`<operand>` is a column

`<value>` is a literal value

The SI algorithm uses a set of known association rules.

S is the number of known rules

R_k represents a rule where $R_k \in \{R_1,\ldots,Rs\}$

k represents a variable with values *1* to *S*

ant represents an antecedent of a rule, R_k

cons represents a consequent of a rule, R_k

ant.C_k refers to the column of antecedent of rule R_k

cons.C_k refers to the column of consequent of rule R_k

$R_k.ant.C_k.V_k$ (or simply $ant.V_k$) refers to the value used in R_k's antecedent

$R_k.cons.C_k.V_k$ (or simply cons.V_k) refers to the value used in R_k's consequent

Two auxiliary functions are used:

Add_pred(*P_i*): this changes the query by adding an additional predicate P_i to the query's `predicate_list`.

Replace_pred(*P_i*, *P_j*): this changes the query by removing the predicate P_i and adding the predicate P_j to the query's `predicate_list`.

The ceiling function $\lceil\ \rceil$ is also used: $\lceil X \rceil$ is the smallest integer greater than or equal to X.

4.2.2 Query used for Semantic Inequivalence Algorithm

The query, *Q*, is used to demonstrate the SI algorithm subroutines defined in Section 4.2.3:

```
SELECT DISTINCT a
FROM   table_t
WHERE column_i <operator> value_i
AND   column_j <operator> value_j
```

This is the minimum query structure required to exemplify the SI algorithm subroutines. This is because 2 predicates are required for incoherence detection. Otherwise only 1 is needed.

The example query, *Q*, is based on *table_t.*

Let table *table_t* have columns *a, b, c, d, e, f*

There is an B+ tree index on columns (*a,b*), a B-tree index on column (*c*) and a B-tree index on column (*e*).

Let table *table_t* have *n* rows. Each page holds *r* rows. So the number of pages is $NPAG = \lceil n/r \rceil$

Let the rules exist:

`If b = value_1 then a = value_2` (75% confidence)

`If b = value_1 then f = value_2` (75% confidence)

`If d = value_4 then e = value_5` (70% confidence)

4.2.3 The Semantic Inequivalence Algorithm

This section defines the SI algorithm, using the notation defined in Section 4.2.1. Following the definition of each subroutine, the subroutine is demonstrated against the sample query and predicates.

Input: Query, *Q*

Output: Transformed Query, *Q′ UNION Q″* where *Q′* is the semantic inequivalent query and *Q″* is the query corresponding to information known from association rules where the rule that is used has less than 100% confidence. Where the used rule has 100% confidence *Q′* is output only since this is semantically equivalent to *Q*. Where a rule covers a query, only *Q″* is output. Null is output in the case of the algorithm determining that a result set is not possible.

Even though the input to the SI query is a conjunction of simple predicates, the output may include a disjunction of predicates.

The algorithm steps through each predicate of a query, that is each simple expression within the *where* clause of an SQL statement. If the column used in the predicate is also the antecedent of a rule, then the rule may be useful to help answer the query.

In the SI algorithm, it is seen that the main components of the control region are *decision making* and *transformation* of the query. The decision making, shown as the *IF* statements, determine the transformation that will be made to the query by the algorithm.

FOR EACH predicate $P_i \in \{P_1,\ldots,P_N\}$*, in Q:*

BEGIN

> /* depending on the operator value apply one of these functions if criteria specified are met. These will increase the potential for using SI by replacing inequality with equality */

IF (P_i.operator = '<>') /* NOT EQUAL */

BEGIN
    Execute inequ_operator
END

IF (P_i.operator = '>')

BEGIN
    Execute greater_than_operator
END

IF (P_i.operator = '<')

BEGIN
    Execute less_than_operator
END

IF ($C_i = ant.C_k$ for some $k \in \{1,\ldots,S\}$)
    /* predicate column is same as the antecedent column in 1 or more rules */

BEGIN
    /* choose a rule. This calls the subroutine to select a rule for applying to the query to create the SI query */

    R_k = *rule_selection()*
END

ELSE
BEGIN

    Output Q

    EXIT algorithm /* since no further rule transformation can take place */

END

IF (select_list =cons.C_k OR select_list= (cons.C_k , ant.C_k))

BEGIN
    Execute select_col_in_rule
END

IF (cons.$C_k \notin$ select_list)

BEGIN
Execute select_col_not_in_rule
END

IF (cons.C_k = C_j for some $j \in \{1,\ldots,N\}$)
/* if the rule's consequent.column is equal to the column of a different predicate in the query */

BEGIN
Execute rule_alt_predicate
END

END

Definition of inequ_operator

IF (C_i = ant.C_k for some $k \in \{1,\ldots,S\}$
AND C_i.number_of_distinct_values = 2
AND C_i.distinct_values = {V1, V2}
AND C_i.V_i = V1)

BEGIN

$P_{(N+1)=}$ (C_i = V2)

Replace_pred (P_i,$P_{(N+1)}$)

/* this will replace a <> SARG with =. This is useful if there is a composite index */

END

End of definition of inequ_operator

Example of *inequ_operator*:

Using the predicate:

```
b <> value_0
```

by substituting for *Q's* predicate:

```
column_i <operator> value_i
```

If there are only 2 distinct values for *b*, then the condition C_i.*number_of_distinct_values = 2* is met.

Additionally, column *b* is the antecedent of a rule, hence the criterion C_i = *ant.*C_k is also met.

If the only other value for column *b* is *value_1*, then the predicate

```
b <> value_0
```

is replaced with:

```
b = value_1
```

If there are more than 2 distinct values for *b*, then the condition C_i.*number_of_distinct_values = 2* is not met. Hence no transformation takes place.

End of *inequ_operator* example.

Definition of greater_than_operator

IF C_i = *ant.*C_k *for some* $k \in \{1,\ldots,S\}$
AND $C_i.V_i = V1$
AND $\{x \mid x \in C_i.distinct\_values \text{ AND } x > V1\} = \{V2\}$

BEGIN

$P_{(N+1)} = (C_i = V2)$

Replace_pred (P_i,$P_{(N+1)}$)

/* if there is a rule where *ant.*$C_k.V_k$ = the greater value , then it will be able to be used for partially answering the query and forming a SI query */

END

End of definition of greater_than_operator

Example of *greater_than_operator*:

Using the predicate:

```
b > value_0
```

by substituting for *Q's* predicate:

`column_i <operator> value_i`

If there is only one value for *b* which is greater than *value_0*, then the last condition is met.

Additionally, column *b* is the antecedent of a rule, hence the criterion $C_i = ant.C_k$ is also met.

If the only value for column *b* that is greater than *value_0*, the value specified in the predicate, is *value_1*, then the predicate

`b > value_0`

is replaced with:

`b = value_1`

If there is more than one value for *b* which is greater than *value_0*, then the last condition is not met. Hence no transformation takes place.

End of *greater_than_operator* example.

Definition of less_than_operator

IF $C_i = ant.C_k$ *for some* $k \in \{1,\ldots,S\}$
AND $C_i.V_i = V1$
AND $\{x \mid x \in C_i.distinct\_values \text{ AND } x < V1\} = \{V2\}$

BEGIN

$P_{(N+1)} = (C_i = V2)$

Replace_pred $(P_i, P_{(N+1)})$

/* if there is a rule where $ant.C_k.V_k$ = the lesser value, then it will be able to be used for partially answering the query and forming a SI query */

END

End of definition of less_than_operator

Example of *less_than_operator*:

Using the predicate:

`b < value_1`

by substituting for *Q's* predicate:

`column_i <operator> value_i`

If there is only one value for *b* which is less than *value_1*, then the last condition is met.

Additionally, column *b* is the antecedent of a rule, hence the criterion $C_i = ant.C_k$ is also met.

If the only value for column *b* that is less than *value_0*, the value specified in the predicate, is *value_1*, then the predicate

`b < value_1`

is replaced with:

`b = value_0`

If there is more than one value for *b* which is less than *value_1*, then the last condition is not met. Hence no transformation takes place.

End of *less_than_operator* example.

Definition of rule_selection

/* *rule_selection* returns the rule to be applied to transform the query. This is the rule most likely to enable an improved data access path to answer the query. */

IF rule exists where cons.C_k has B+ tree index
BEGIN
    return R_k
END

ELSE

BEGIN
    IF rule exists where cons.C_k has B tree index

    BEGIN

IF more than 1 rule
choose rule with highest confidence : return R_k

END
ELSE / no indexes */*
BEGIN

choose rule with highest confidence : return R_k

END
END

End of definition of rule_selection

Example of *rule_selection:*

Using the predicate:

`b = value_1`

by substituting for *Q's* predicate:

`column_i <operator> value_i`

Column *b* is the antecedent of 2 rules:

```
If b = value_1 then a = value_2 (75%) and
If b = value_1 then f = value_2 (75%)
```

The consequent of the first rule has a B+ tree index. Hence *rule_selection* will choose the first rule. Output of *rule_selection* is the rule:

```
If b = value_1 then a = value_2
```

End of *rule_selection* example.

Definition of select_col_in_rule

IF (R_k.confidence = 100%)

BEGIN
IF {P_i } = { P_1,..,P_n } / there is the only 1 predicate in Q */*
BEGIN

Q is answered completely by cons.V_k /* rule covering */

Q'' = select cons.V_k /* Sybase allows *select* of a literal without a *from* clause */

Output Q''

EXIT algorithm

END

ELSE
BEGIN

$P_{(N+1)}$ = (cons.C_k)

Q' = Add_pred($P_{(N+1)}$) to Q

Output Q'

EXIT algorithm

END

END

ELSE / confidence < 100% */*

BEGIN

Q is answered partially by cons.C_k

IF (cons.C_k has an index)

BEGIN

$P_{(N+1)}$ = (cons.C_k < cons.V_k OR cons.C_k > cons.V_k)

use < and > rather than <> /* the query will no longer request what is known from the rule */

Q' = Add_pred($P_{(N+1)}$) to Q

END

ELSE / no index on cons.C_k */*

BEGIN

$P_{(N+1)}$ = (cons.C_k <> cons.V_k)

Q' = Add_pred($P_{(N+1)}$) to Q

END

/ add the 'known' part of query – known from the rule */*

Q'' = select cons.V_k

Output Q' UNION Q''

EXIT algorithm

END

End of definition of select_col_in_rule

Example of *select_col_in_rule:*

The select list of *Q* is column *a*. This is also the consequent of the rule output from *rule_selection.* The rule has less than 100% confidence hence the *else* part of the subroutine will be executed. The consequent of the rule *a = value_2* is the partial result set.

$P_{(N+1)}$ is the negation of the rule's consequent, that is:

```
a <> value_2
```

Since the consequent column is indexed, $P_{(N+1)}$ will be:

```
(a < value_2 or a > value_2).
```

This is union-ed with the known part of the query that is the rule's conseqent.

However, if the rule did have 100% confidence and the query *Q* only had the 1 predicate: `b = value_1`, then the rule would answer the query, with the consequent `a = value_2` being the result set.

End of *select_col_in_rule* example.

Definition of select_col_not_in_rule

IF (R_k.confidence = 100%)

BEGIN

$P_{(N+1)}$ = (cons.C_k)

Q' = Add_pred($P_{(N+1)}$) to Q

Output Q'

EXIT algorithm

END

End of definition of select_col_not_in_rule

Example of *select_col_not_in_rule:*

Let the select list of query *Q* be column *c* instead of column *a.*

If the rule has 100% confidence then the predicate

```
a = value_2
```

is added to the query.

End of *select_col_not_in_rule* example

Definition of rule_alt_predicate

IF (R_k.confidence = 100%)
BEGIN

IF (C_j.V_j <> cons.C_k.V_k)

BEGIN

Q has no result set

Output NULL

EXIT algorithm

END

END

End of definition of rule_alt_predicate

Example of *rule_alt_predicate:*

Using the predicate, P_j

`a = value_3`

by substituting for *Q's* second predicate:

`column_j <operator> value_j`

If the rule has 100% confidence, then there is a null result set since the query is contradicting a rule with 100% confidence.

End of *rule_alt_predicate* example.

4.3 Conclusion

This chapter has defined and demonstrated the SI algorithm in detail. A sample query was defined that was used to exemplify each module of the SI algorithm and how it transformed the query.

Chapter 5

Cost Comparison

5.1 Introduction

Chapter 4 specified and exemplified the SI algorithm in detail. To build upon this, this chapter looks at the cost of using SI because for SI to be a useful query processing technique, it should be able to reduce the cost of answering queries in some definable situations. There is no single solution that is ideal for every possible scenario. Hence SI should be an efficient method in some identifiable circumstances.

In Section 1.2, we discussed that the motivation for this thesis is to increase the usage of association rules by employing them to partly execute a query. The rules are taken advantage of by using them to re-write the query into one that is semantically different to the original query - hence taking the information from the database rules into account. This chapter demonstrates that by using SI in this way it can reduce the cost of answering queries because it reduces the requested information, consequently making the result set narrower due to requiring less data to be retrieved. This is efficient if it results in lower query processing cost. It is seen that the response time for answering the SI query may be less than that for answering the original query, given by the potential of being able to use a more efficient data access path that would not have been used by the original query but is suitable for the modified query.

For the purpose of demonstrating the effects of applying the SI algorithm on query processing cost, this chapter compares the cost of answering queries in their original form, with the cost of using the corresponding SI queries. This is done in order to help identify and establish the situations where SI adds the greatest advantage. The effect on the cost of processing a query is examined by using sample database queries, based on the data model defined in Section 3.8. Each query is walked through the SI algorithm. The cost of the original query, before applying SI, is compared with that of the SI query, which is achieved as the resultant output of the SI algorithm. Hence the effect of SI on query cost is demonstrated.

To further and independently demonstrate the benefits that SI has on query processing costing, Appendix A5 describes a formal query processing algorithm known as the *Decomposition Algorithm* [50]. This is used for autonomously demonstrating how SI can reduce the query processing cost in a generic way, without using any specific query example.

5.2 Overview of Semantic Inequivalence in Action

This section gives an overview of how the application of the SI algorithm can help process a simple query. For this, we use a query based on the table, *table1*.

Let table *table1* have 4 columns, *a*, *b*, *c* and *d*, each of equal data type and length, with a composite index on columns (*a*, *b*).

Say *table1* has 1,000,000 records stored on 100,000 pages. Hence a full table scan would require 100,000 I/Os – that is 1 I/O per page.

Example 5.1:

A query, *Q*, is:

```
SELECT DISTINCT a
FROM   table1
WHERE  b = value_1
```

Let the rule:

```
if b = value_1 then a = value_2
```

exist with 75% confidence.

With SI application, the query *Q* would be changed to *Q'*:

```
SELECT  DISTINCT a
FROM    table1
WHERE   b = value_1
AND    (a > value_2
OR      a < value_2)
```

The SI query, *Q'*, takes into account that *value_2* must be in the result set. This is given by definition of the rule, and hence can be eliminated from being requested by the query. It is removed from being requested by the extra specification via the addition of predicates to only retrieve column *a* where the value is not equal to *value_2*, the value known or given from the association rule.

The additional query conditions may encourage use of the index for two reasons.
- With the new predicates that are added to form query *Q'*, the optimiser would be more likely to use the index, since there is an increase in selectivity.
- Moreover, the leading column of the index is being used for *Q'* as a predicate in the query, giving a key column starting point for an indexed-based search.

Say for example, column *b* has *value_1* for 20% of the rows (200000 rows), then given the association rule, we know that column *a* has the value of *value_2* for 75% of these 200000 records. Using the index this would involve reading only index pages - no data pages would need to be read because the index covers the

query, and only a subset of the index pages would need to be read – that is, those where (*a* > *value_2* or *a* < *value_2)* and *b* = *value_1*.

If the index size is 50% of the table size - assuming equal sized columns, then this would result in 50000 pages being read if the whole index is scanned. However, with the SI query, if approximately 75% of 20% (where *b* = *value_1*) of the index pages are read, then only 7500 pages would be read plus 1 rule page, assuming there is a single rule page.

If an average index is 10% of table size [45], then by SI enabling index usage it will reduce I/O to approximately 10% of what it would have been previously, with the original query.

If the rule has 100% confidence then only *nrp* I/Os would be needed, where *nrp* is the number of rule pages. This compares to 100000 I/Os for a full table scan. This is because the rule would answer the query completely – the situation of *rule covering*.

If however the rule's confidence is 75% as given, and there is no index on the columns (*a,b*), then using SI would actually be more expensive, because it would still require 100000 I/Os for the full table scan, plus *nrp* I/Os for reading the *rules* page. In this situation, the original query should not be changed.

This demonstrates a crude, yet common style of query that can benefit from the application of SI. It aims to provide an explanation via an example of the advantage of using the SI process and its effects. It is extensible for more complex queries, as will be seen in the examples throughout Chapters 6 and 7.

5.3 Comparing Costs

This section compares the cost of using SI to not using SI, with both B-tree indexes and bitmap indexes. The comparison is based on the query example in Section 4.2.2:

```
SELECT  DISTINCT a
FROM    table1
WHERE   b = value_1
```

Costs are based and compared on the I/O required to answer the query. I/O is used instead of elapsed time or CPU time for several reasons: it is consistent across query executions, it is not dependent on other processes running on the machine and it is independent of how slow or fast the underlying hardware may be. Hence it is more reliable.

5.3.1 With B-Tree Indexes

Given the original query, *Q*:

```
SELECT  DISTINCT a
FROM    table1
WHERE   b = value_1
```

If there is no index on column *b*, this would require a full table scan, which is *NPAG* I/Os, as defined in Section 4.2.2.

With a composite index on columns (*a*,*b*), the original query *Q* cannot make use of the index since *Q's* predicate operand is not the leading indexed column, *a*. However, an SI query, *Q'*, derived from *Q* can make use of the index as now discussed.

The SI query, *Q'* , is:

```
SELECT DISTINCT a
FROM  table1
WHERE b = value_1
AND  (a > value_2
OR    a < value_2)
```

This is formed because the original query can be answered partly by the first of the three association rules defined in Section 4.2.2 namely,

`If b = value_1 then a = value_2 (75% confidence)`

and can therefore be altered accordingly by application of the algorithm.

The resulting SI query *Q'* can make use of the index because it has a predicate within its predicate list whose operand is the leading indexed column, *a*.

The I/O required for the SI query would be:

number of data pages to read while using the index
+ number of rule pages to read
+ number of index pages to read

We consider these below.

IF:
(number of index pages to read + number of rule pages)
is less than
(NPAG – number of data pages to read while using index)

THEN:
The application of SI has resulted in less I/O to answer the query.

This condition reflects the difference between executing the SI query, *Q'*, with the original query, *Q*, in terms of the difference in the I/O cost that would result.

Number of Data Pages

The number of data pages that must be read may be significantly reduced given the information in the rule relevant to the original query. If half of the rows in *table1* have *b = value_1*, then since the confidence of the rule is 75%, approximately 25% of those rows, that is 12.5% of the rows in the table as a whole, do not have *a = value_2* and therefore can satisfy the predicate *(a > value_2 or a < value_2*). The query only needs to access those rows. The exact number accessed may be less that 25% of the rows with *b = value*_1: for example, null values would not satisfy the predicate although not being equal to *value*_2 and so between 0% and 25% of the rows with *b = value_1* would need to be accessed.

Number of Rule Pages

To enable SI to be efficient, the number of rules and hence rule pages should be limited to those that are most useful to the database querying patterns, coupled with those that are most powerful or appropriate. This means that the stored association rules should be of relatively high relevance, such that they can be used for *ad hoc* queries in a very large database, or a data warehouse, and significantly improve response time. This is the pre-requisite of the association rules that should be selected for storage. For example, a rule that has a high confidence level but is not providing information that is relevant to the database querying should be filtered out. Minimising the number of rule pages is important to minimise the impact on I/O. The greater the number of rule pages, then the more read I/Os are required, offseting the advantage of using SI.

Such a set of ‘most relevant rules’ can be determined by a database administrator or database specialist with local knowledge of data usage. The criteria used for selecting rules should be based on criteria such as:

- how frequently the attributes in the rule are queried. For example if a rule's attributes are never queried it may not be relevant to the applications
- high confidence level of rule. The greater the rule's confidence the more impact it has on query selectivity when a query is changed to use it.

The most 'powerful' or 'relevant' rules refer to those association rules that help reduce the query processing cost by changing the data access path. This includes those rules that:

- enable rule covering
- enable an index to be used
- enable an index to be 'better' or more efficiently used
- enable incoherence detection.

Number of Index Pages

The depth of an index or number of index levels is a direct measure of performance of using an index in evaluating a query [24].

The number of index levels to traverse through depends upon the number of rows in the table and the length of the indexed key / columns. The number of index levels can also be affected by fragmentation in the database. Therefore the number of index levels can be hard to predict precisely. However, using a RDBMS, such as Sybase Adaptive Server Enterprise, a table structure such as *table1* with 1 million records and an index on *(a,b)* has 3 levels in the index (determined using a Sybase system command, *sp_estspace*), and *NPAG* is 16000.

Substituting this in the above cost comparison formula:
(number of index pages to read + number of rule pages) < (NPAG – number of data pages to read while using index)

The number of rule pages should be as small as possible since more rule pages

will progressively provide less and less relevant rules, whilst requiring more I/O. Hence assuming a single rule page and substituting 1 in the formula for *number of rule pages,* we have:

If *(3 + 1) < (16000 – number of data pages to read using index)*
then the query transformation will have been effective.

Given that *NPAG* is the total number of datapages,
number of data pages to read while using index will be approximately:

*percentage of rows that need to be read * NPAG*

Hence the more selective the query becomes the *percentage of rows that need to be read* becomes less, reducing the *number of data pages to read using index.*

This being true is likely with effective indexing. Given the *number of index pages to read* is relatively small [45], rarely above 3, we can safely assume the reduction in I/O attributed to SI application as being the reduction in the number of data pages to read using the index.

In this scenario, if 12.5% of rows need to be read and assuming that the index is clustered, then the number of data pages to read using the indexed access would be 12.5% of 16000 pages, or 2000 pages. Substituting this in the SI cost reduction formula:
(3 + 1 + 2000) < 16000.

Therefore with SI, there is a reduction in I/O of 13096 or nearly 81%.

If the original query could have used the index, such that SI would not change the access path, then there may still be some advantage in using SI. This would be the case if it adds a predicate on a column that is also included in the index because this would increase the data selectivity. However, the advantage would not be as profound as when SI enables the access path to be changed from a full table scan to indexed access as in the example.

If the index were on the same columns but in the reverse order, (*b*,*a*), such that it could have been used by the optimiser as the access strategy before applying the SI algorithm, then there may still be some cost advantage in using SI. The advantage would not however be as prominent as when SI enables the index to be used when otherwise a full table scan would be used.

If the index were on columns (*b*,*a*), the original query would be able to use it, as the predicate's column *b* is on the leading column of the index, providing a starting point for an index-based search. In this case, the number of pages accessed would be:

number of index pages to read + number of data pages to read

If 10% of the rows have *b = value_1*, then 10% of data rows would need to be accessed. If additionally, SI introduces the clause of *a <> value_2*, and 75% of rows have *(a = value_2* where *b = value_1),* as the rule stipulates, then only 25% of the 10% of the rows need to be accessed with SI application. Therefore, using SI to change the query, rather than 10% of total rows being accessed in this case, only 10% * 25% of rows would need to be accessed, or 2.5%. This is approximately 2.5% of the data pages rather than 10% of the total data pages – a 75% reduction. Given the *number of index pages to read* is relatively small [45], rarely above 3, we can safely assume the reduction in I/O can be attributed to SI application.

Generally, rather than *x*% of rows being accessed, where *x* is the selectivity of the leading index column that the query would be able to use, (100 – *y*)% * *x*% of rows would need to be accessed, where *y* is the confidence of the rule. This results in only (100 – *y*)% of the number of rows that would otherwise be retrieved. Hence the higher *y* is, or the higher the rule's confidence, then the more selective the SI query becomes – resulting in greater impact using the same query plan.

5.3.2 B-tree (Non-Clustered) Indexes vs B+tree (Clustered) Indexes

Clustered indexes work well with predicates that include a range, such as, *less than* < and *greater than* >, and therefore with SI. This is because the existence of rules, upon which SI is based, implies that values are repeated. Hence clustering them, or physically storing them together, can help process a query more efficiently, since more 'relevant' data is read from a page compared to randomised data ordering.

Non-clustered indexes are generally more suitable for highly selective point queries or where less than 10% of rows would be retrieved [32].

For example, a query *Q* has a predicate on column *b* with 30% selectivity of the table's records. Even if column *b* has a non-clustered index on it, then a full table scan is likely to be chosen as the access strategy by the optimiser rather than the non-clustered index, because of the high range of values that need to be accessed that are not physically stored together. Rather they are disparately stored among different pages, unlike with a clustered index storage strategy.

Non-clustered indexes can be useful with SI if previously the index would not be used by the original query due to it not being selective enough. The chance of a more efficient access path is likely to arise if SI creates either a point query or a significantly more selective query, from what previously was not selective enough, or not including the leading column of an index. If none of these are case, then SI is unlikely to improve the cost of processing the query.

If, however, predicates with < and > are added, where the predicate column is indexed, or is the leading column of a composite index, then SI is less likely to change a query plan if the predicate does not filter a significant proportion of rows. Full table scanning may be more efficient than using a non-clustered index where data is not physically in order and a significant proportion of pages need to be accessed. This is because if most rows need to be read then indexed access for a large number of rows can result in more I/O than accessing the entire table just once with a full table scan. Full table scanning also uses sequential access, which

in practice, with multiple block reads and large I/O sizes, increases the efficiency of this type of access.

Non-clustered indexes may help in conjunction with SI for rules with less than 100% confidence and where even adding an inequality clause (for example, *a* > *value_2 or a* < *value_2*) is selective enough for non-clustered indexes to be preferred over other access strategies.

A unique usage of non-clustered indexes is their ability to completely answer a query, known as *index covering*. This is where the index has sufficient information to answer the query without requiring access to the data pages. Only non-clustered indexes have this ability because they are dense indexes. This means that they have an entry for every data row, unlike clustered indexes, which have an entry for the first row of each data page.

If SI enables index covering to be used, then the advantage can be higher than when it enables a clustered index to be used.

For clustered and non-clustered indexes, the principles are similar, except that increasing selectivity is not as important with clustered indexes. Additionally, the benefits can be achieved even when the query is a range query. SI will be beneficial with both clustered and non-clustered indexing if either:

- SI enables an index to be used by adding the leading column of the index to the query
- Selectivity is increased to improve the filtering of the query result set higher up the index levels.

If the index is clustered, then using < and > in a new predicate will generally be more valuable than using it in a non-clustered index, as the data in a clustered index is physically in the indexed attribute order.

5.3.3 With Bitmap Indexes

Even though the cost saving criteria, as defined above, are especially useful with

'index enabling', it has so far only been discussed with the variants of the B-tree index structure. However, given that SI is suitable for very large databases, such as data warehouses, which bit-mapped indexes are very much oriented at [22], the use of bitmap indexes should be looked into in conjunction with SI as they can complement each other.

Both bitmap indexing and SI are especially beneficial for *ad hoc* queries and very large databases with low levels of data modification transactions. Hence they can be used complementarily.

Bitmap indexes can give huge performance gains even on the lowest end of hardware and reduce response time dramatically for some *ad hoc* queries [32], if they can be used.

Bitmap indexes are most effective for low cardinality columns (such as *marital_status*). These types of attributes often have rules intuitively associated with them, and can be highly efficient for tables with many rows as would be found in a data warehouse type scenario. When bitmap indexes are used for columns where each value is repeated many times, the bitmap index will typically be less than 25% of the size of a regular B-tree index [22].

If the use of a bitmap index for query optimisation is enabled by SI application or at least further improved by promoting selectivity, then I/O can be significantly reduced.

For example, given the query:

```
SELECT DISTINCT a
FROM   table1
WHERE  b = value_1
```

With SI, the query becomes:

```
SELECT DISTINCT a
```

```
FROM   table1
WHERE  b = value_1
AND    a <> value_2
```

Bitmap indexes can be used as effectively for inequality (<>) as they can for equality (=). This is demonstrated below using the NOT operator.

If there are 5 distinct values for column *b* and 3 distinct values for column *a*, an example bitmap index on column *a* is:

```
a = value_1    a = value_2    a = value_3
1              0              0
0              1              0
0              1              0
0              0              1
```

Each row represents a row in the actual table. A *1* bit indicates the row has the value. A *0* bit indicates that it does not have the value.

Using bitmap indexes for the query:

```
b = value_1    AND  NOT  a = value_2
1              AND  NOT       1         =         0
1              AND  NOT       0         =         1
0              AND  NOT       0         =         0
0              AND  NOT       1         =         0
```

This shows that only the second row meets the criteria of the *where* clause, with a result bit of *1*, or *true*.

If bitmap indexes are used correctly, on columns with the same value repeated many times, which is complementary to SI, because association rules also require

values to be repeated to give data patterns - then they typically take less than 25% of the size of a regular B-tree index. Therefore storage can be very compact, consequently requiring less I/O for accessing them.

Let table *table1* have 1million rows with column *b* having 3 distinct values and column *a* having 5 distinct values.

Using bitmap indexes on these would require:
*1000000 * 8 bits = 1 million bytes* or 1MB to store the index.

The cost advantage of using a bitmapped index in this way depends upon whether it would have been used before SI was applied to the query, or if SI enables it to be used when it would not have been used otherwise.

If a bitmapped index is used after SI is applied only and would not have been used previously, then the cost of the SI query using the bitmap index is:

number of bitmap index pages to read + number of rule pages to read

If this is less than *NPAG*, then SI is useful in reducing query cost.

The number of bitmap index pages is *x* MB, where *x* is:
(number of rows) + (number of distinct values in column) = n bits.
*(n bits / 8) / (1024 * 1024) = x MB.*

If the bitmap index could be used first before SI, then adding SI will increase selectivity, hence making bitmap index use more effective in reducing the number of rows returned.

Taking the original query, *Q*, for example,

```
SELECT DISTINCT a
FROM   table1
WHERE  b = value_1
```

Only the bitmapped index on column *b* would need to be accessed. If this has 3 distinct values, then 1000000 * 3 bits would need to be accessed, resulting in accessing approximately one-third of the table's records assuming uniform data distribution.

Now, with the SI query, rather than selecting one-third of the rows, it only selects (1/3 * selectivity of new predicate) of the table rows.

The higher the selectivity of the additional predicate is, the fewer the number of rows that need to be accessed.

Hence, SI can reduce the number of pages that need to be read at the table level by the selectivity of the predicates that are added as a result of applying the SI algorithm.

5.4 Exceptional Cases

The above cost criteria for success are applicable if a relevant rule has less than 100% confidence. There may however be some cases where a rule has 100% confidence, and can *cover*, or completely answer, a query. The query that can be answered using this rule would have I/O reduced to the number of rule pages in the database. If there exists only 1 rule page, then only 1 I/O would be required to answer the query. Undoubtedly this would be more efficient than any other access strategy, such as a full table scan or an indexed access path.

The rule page is likely to be cached if it is used frequently for the SI process, using a least-recently-used cache page-ageing algorithm. Alternatively, it can be pinned to a page in memory to prevent the need to read it from disk.

Rule covering can be a useful and powerful access strategy in a very large

database that has some data patterns that are otherwise unknown to the DBMS (without the searched rules) and hence would otherwise require a large amount of resource to execute.

Example 5.2:

```
SELECT no_of_staff
FROM stores
WHERE store_name = 'The Bookshop'
```

This has the relevant rule:

```
If store_name = 'The Bookshop' then no_of_staff = 6
(100% confidence)
```

This rule answers the query, not requiring access to the data or index pages.

There is also the opposite exceptional case where the SI algorithm may discover a query has no result set via incoherence detection (the last part of the SI algorithm). This is efficient compared to finding an empty result set by reading a table or index, as this can be found with the SI algorithm, with minimum computational cost, if any I/O, in particular since the rule page, is cached.

5.5 Conclusion

This chapter has exemplified the effects of SI on the cost of processing a query.

A query is appropriate for SI application if there is a useful rule that can be applied. A useful rule is one that can add predicates to a query, such that the addition of the predicates enables an access path to be chosen by the query optimiser that results in less I/O.

Generally, SI can reduce the cost of processing a query in situations where the access path used by the optimiser for the SI query is more efficient than the path that would be used for the original query, holding other factors constant. However, even with the same access path (if an indexed path) being used for the original and SI query, the latter may still be more efficient, but the improvement will not be as significant. If full table scan access is used before and after, then there will be no gain to using SI.

The cost will be more reduced if there is an improved physical data access path, such as a useful index which can be used after SI is applied, which would not have been used otherwise, either because without the additional column the query would not be selective enough or because the leading part of the index was not being used before SI. I/O will be reduced by the number of pages read for a full table scan, minus the number of pages read using an alternative access technique, plus *nrp* rule pages.

A generic style query was used to show the cost comparison with both B-tree and bitmap indexes. This was subsequently mapped to specific queries. As demonstrated by the examples, SI will give the greatest advantage where adding a query predicate enables the use of an index where a full table scan would have been used otherwise.

SI also adds a big advantage where either a rule covers, or answers, a query or where a rule tells that there is no result set, because both of these scenarios can prevent an otherwise expensive operation, such as a full table scan, from taking place.

Appendix A5 demonstrates cost comparison of queries before and after applying SI using an independent costing technique with the established QUEL Decomposition Algorithm, but replacing QUEL with the SQL query language.

Chapter 6

Real-world Examples

6.1 Introduction

Chapter 6 puts the SI algorithm to practical use by applying it to tackle real world queries, hence providing empirical evidence of its usefulness. For this purpose, two completely separate and independent real-world databases are used. Two databases are used to strengthen the evidence for the usefulness of SI. Also, the effect of possible bias existing in one database and hence arriving at skewed conclusions is reduced.

First of all, a set of rules was manually derived from each database. The rules have varying degrees of confidence – some below 100% and a few at 100%. The cross section of queries executed is selected to show cases where the application of SI provides huge advantages to cases where there is no improvement at all with the application of SI.

Prior to the demonstration and use of SI against the real database queries, this chapter provides some background on the choice of databases, rules and queries that are used. This is followed by an overview of the information produced by the DBMS optimiser's query plan and what it means. Then the actual queries are listed.

In Sections 6.5 and 6.6, for the real-world queries, each query example set lists the I/O costs, the original query followed by appropriate association rules if any,

and the corresponding SI query. Each 'query set' (consisting of an original query plus its corresponding SI query) is compared in terms of the I/O required for each – showing the situations where SI gives benefit and how significant the benefit actually is. The detailed query plans chosen by the optimiser are listed in Appendix A4. The results and findings are summarised in the following section.

6.1.1 Motivation of Research Method

Six main categories were identified in terms of the effect of SI on query processing cost. Each query fell into one of these categories. These categories are listed in Table 6.1 along with the impact on I/O and the query examples that exemplify them.

Seeing the impact on I/O reduction that SI can have helps motivation for the study of SI queries and their comparison with original queries as conducted in Section 6.5 and Section 6.6.

Cause of change between original query and SI query	**Average approximate change in I/O**
SI answered by rule (either rule covering or incoherence detection) – requires 100% confidence rule (6.3, 6.8, 6.15)	4000 times less
100% confidence rule that does not answer SI query but enables index access (6.6)	1000 times less
SI enables index covering (6.2, 6.11, 6.15)	75 times less
SI enables use of index (6.1, 6.5, 6.7, 6.12, 6.16, 6.17, 6.18, 6.19)	24 times less
SI improves selectivity – but same access path (6.9, 6.10)	4 times less
No change in access path (6.4)	1.5 times more

Table 6.1 – Categorisation of Query Processing with SI

This shows that SI may be a very powerful query processing technique in several types of situation. The most powerful usage is in the situation where I/O is reduced to 1 via rule covering. Substantial degrees of improvement are also seen where a more efficient access path is made available to the query optimiser following the application of the SI algorithm. Section 6.7 discusses the results in more detail.

The graph below compares the I/O for each query example.

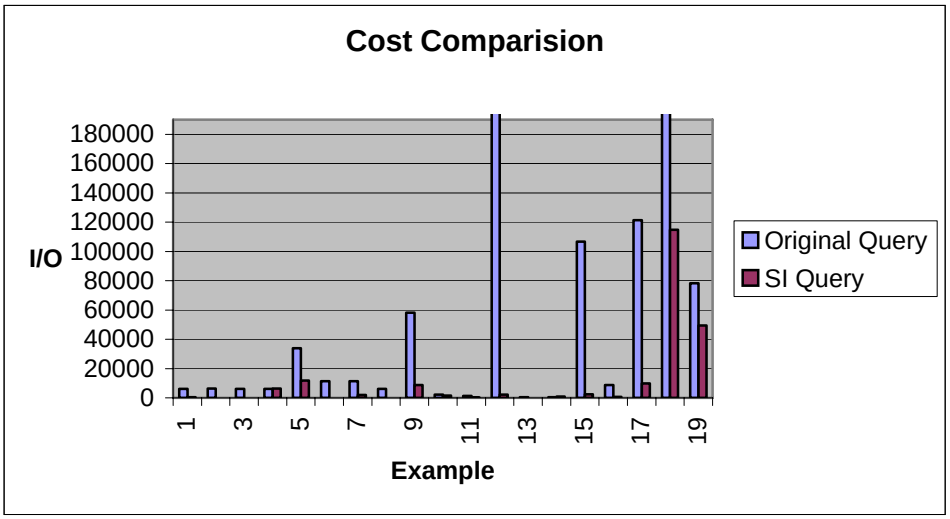

Figure 6.1 – Cost Comparison With and Without SI

6.2 Background and Reasons for the Choices

6.2.1 Choice of Databases

For the empirical analysis of SI, two large real-world databases were used, running on Sybase Adaptive Server Enterprise.

The first database is a 14 gigabyte (Gb) management information system (MIS) database used to derive a set of sample rules and execute queries against. This

database was chosen for a number of reasons. Firstly, it is the type of database that SI has the most potential value added for. This is because it is a large data warehouse type database with relatively static data used for a decision support system (DSS) and MIS type applications and queries. As highlighted throughout Chapter 2, this is the type of database system that SI is ideal for. This MIS database is actually used for reports and *ad hoc* queries for historical information on stocks and asset portfolios. It is not updated frequently and is used to provide a data warehouse style database for a host of asset management business applications. It is also relatively large in size, and there are not many such large real world databases that one has access to, and are simultaneously suitable for this purpose.

The second database used for the set of real world examples is a 9Gb online trading database with live data feeds. Some tables were holding many millions of records. However, despite updates to the data, patterns remained similar, hence rules were found to exist. With respect to association rules, the database was therefore relatively static. The type of data that was changed were attributes like security prices – whereas rules existed on other non-price attributes, which are queried for reporting purposes, including regulatory reporting.

The SI algorithm was applied to each query against its respective database and the costs, based on I/O, of the original query and SI query were compared.

The sets of sample queries were manually extracted from the applications that use the databases. The queries were found either embedded in the client applications or within stored procedures that the clients execute. The selection of queries aim to show where SI is most useful, and where it is not useful. A wide cross-section of queries was taken with the intention of providing a varied sample, yet choosing queries that 'fit' into the SI framework in terms of having relevant rules that can be used to help answer the queries and enable transformation. Thus if an association rule existed on column *currency* for example, a query with *currency* as a predicate would be a good candidate for selection. If none of the column attributes used in a query had any relevance to the rules found, then SI is not relevant to use hence such a query was not selected

as it would give no value to demonstrating SI because the SI query would be the same as the original query.

6.2.2 Finding Association Rules

To manually search for rules, a subset of queries were picked from the applications that use the database, and rules based on attributes that these queries used were searched for. Rules were found by searching for patterns among columns of some of the largest tables used by the queries. Columns that hold data items such as *codes* and *indicator* values were predominantly searched as they are likely to have data relationships or patterns more than those highly unique attributes with date/timestamp values, float data types or identifier values (such as primary or alternate keys) which tend to be used for point queries. The searching was done by manually querying the database tables.

Moreover, it was also observed in the applications that queries were often based upon columns that hold *codes* and/or *indicator* values. For example, queries may be based on finding *currency codes* for particular *country codes* or *account types* etc. This type of data is generally useful for DSS type applications or DSS queries. Hence finding rules that SI can use may be very useful for corporate decision system queries and vice versa.

The rules used for query cost comparison have single column antecedents. This is to keep the process of demonstrating SI simple and to the point. Also, the example association rules from researched data mining papers were very much based around single attributes. Hence this was considered sufficient for demonstrating SI.

In the first database, a lot of data was found to be stored on the clients of the organisation – customers or clients being a core part of any company's data. Likewise, it was found that many of the application queries were based on retrieving client information. Hence rules were especially sought for regarding clients. For example, a common query found was based on finding the currency that the client uses as his/her base or main currency for dealing in stocks and

shares. However, given the base currency, there is a high confidence of the client's country of residence being a particular value, and here SI enables the optimiser to take advantage of this sort of relationship.

In the second database, the data stored is based on trades, securities, and curves, which are used for the pricing of trades. Therefore, the queries found were based around attributes of these entities for the second set of real-world sample queries.

The corresponding SI queries were manually generated by stepping the original queries through the SI algorithm.

6.3 Optimiser Plan Explanation

Each query is followed by the I/O statistics required to answer it followed by the I/O needed to answer the corresponding SI query. The detailed query plan or execution path that the optimiser chose to process the original and SI queries is listed in Appendix A4. The statistics for the queries were collected using a Sybase Adaptive Server Enterprise feature to display the query plan chosen by the optimiser (*set showplan on*) and the I/O cost involved (*set statistics io on*).

The core information in the query plan tells us the table/s being accessed, and whether an index or full table scan is being used for the data access. If it is using an index, it specifies which one, and the keys or columns in the index that are being used for the search request. *ASC* stands for an ascending index scan, whilst *DESC* for descending index scan – which is effectively backwards reading of the index.

We can identify index covering by the description '*Index contains all needed columns. Base table will not be read'*.

I/O size can be between 2Kb and 16Kb, and may be different for a table and its index. The buffer replacement strategy refers to the cache replacement strategy if pages need to be read from disk. Pages read from disk can either replace the LRU page in cache or the MRU page in cache.

Dynamic index refers to using an OR or an IN within a query.

Getsorted may be used where for example the *distinct* keyword is used in a query, so that the optimiser needs to sort the rows into a worktable (or temporary holding space) for the operation.

Worktable signifies a table created by the optimiser during query execution, for example, when performing *distinct*, *sort* (for ordering data) or *grouping* operations.

Forward scan means that access is starting at the beginning of the index or first qualifying row, going through subsequent pages by following page pointers to the next page.

6.4 Statistics Output Overview

As well as including the query path that the optimiser chooses for each query, the query optimiser's output also includes statistics on the input and output that is required for query execution.

The statistics section provides information on the number of table accesses, page reads and disk reads that are performed.

Scan count is the number of times that a table is accessed by a query.

Logical reads refers to the total number of reads – from cache or disk.

Physical reads are the number of reads from disk.

APF (asynchronous pre-fetch) refers to the number of reads that the optimiser makes from disk in anticipation of a page being requested, even though it is not needed at the time it is read into cache. The optimiser may decide to bring pages physically located together on disk into cache if it thinks it will be needed for the query. The rest of reads are 'regular'- pages read from disk due to the data on them being requested. Therefore,

Total reads = regular reads + APF reads

Total writes: the number of writes can be greater than 0 for a query if by its reading pages into cache, it causes a dirty (or modified) page in cache to move past a marker that indicates that it needs to be flushed or written to disk.

6.5 The Query Examples – First Data Set

The tables and indexes used for the first data set examples are listed. Following the table name is the list of columns, their datatype and the indexes on the table.

Table: *TClnt*
(ClientId int
ClientType char(4)
Title char(10)
Fname char(40)
Lname char(40)
AddrLine char(200)
CtryResidenceCode int
Profession char(20)
PrInd int
TelNo char(20)
FaxNo char(20)
Email char(40)
LastUpdate datetime
Ccy char(3)
Charity char(1)

Deal char(1)
LocalInt char(1)
Description char(255))
Indexes:
Clnt_x1 (ClientId) – Primary Key
Clnt_x2 on (CtryResidenceCode, Ccy)
Clnt_x3 on (PrInd, Profession)
Clnt_x4 (ClientType)
Number of records: 100000

Table*: tPortfolio*
(PortNo int
PortName char(5)
Description char(255)
Ccy char(3)
Base char(1)
GroupSector int
Industry int
IndexCategory int)
Indexes:
Pf_ix1 (PortNo) – Primary Key
Pf_ix2 (Base)
Pf_ix3 (IndexCategory, GroupSector)
Number of records: 100000

Table: *tCompRet*
(PortNo int
CompositeType int
BalDate datetime
ClassificationCode int
BaseFee char(1)
HurdleRate float
ExpEstimate float
FundingSpread float)
Indexes:
IxCompReturn1 (PortNo, CompositeType) – Primary Key
IxCompReturn2 (BalDate)
IxCompReturn3 (ClassificationCode, CompositeType)
Number of records: 150000

Table: *TQuote*
(SecId int
SecType char(4)
Category char(4)
QuoteCode char(12)
MarketCode char(2)
Bid float
Offer float
Base float)
Indexes:

Tqu_x1 (QuoteCode, MaketCode)
Number of records: 1000000

Example 6.1:

In this example, I/O is reduced by 95%. The original query uses a full table scan while the SI query enables the use of an indexed access because the predicate that is added is on an indexed column.

Original Query:

```
1> select * from TClnt where Ccy = 'ZAR'
```

Rule: `if Ccy = 'ZAR' then CtryResidenceCode = 220 (90% confidence)`

SI Query:

```
1> select * from TClnt where ccy = 'ZAR'
2> and ctrycodeofresidence = 220
3> union
4> select * from TClnt where ccy = 'ZAR'
5> and (ctrycodeofresidence < 220
6> or ctrycodeofresidence > 220)
```

Original Query: 6154 I/Os
SI Query: 364 I/Os

Example 6.2:

In this example, I/O is reduced by 200 times. This is because the query optimiser decides to full table scan for the original query, while the SI query enables an

index to be used which covers the query so that the underlying table does not need to be accessed. The index answers the SI query.

Original Query:

```
1> select distinct PrInd from TClnt
2> where profession = '160'
```

Rule: `if Profession = '160' then PrInd = 'Y' (95% confidence)`

SI Query:

```
1> select distinct PrInd from TClnt
2> where profession = '160'
3> and (PrInd < 'Y' or PrInd > 'Y')
```

Original Query: 6348 I/Os
SI Query: 37 I/Os

Example 6.3:

This example demonstrates the most powerful use of SI – where the rule has 100% confidence and ‘covers’ or answers the SI query without the need to access the table or index. The SI algorithm will answer this without the need for an SI query. There are no indexes on either of these columns.

The original query requires a full table scan. With SI, read of the rule page is required where it is determined that the rule answers the query.

Original Query:

```
1> select distinct Deal from TClnt where Charity = 'Y'
```

Rule: `if Charity = 'Y' then Deal = 'N' (100% confidence)`

Original Query: 6154 I/Os
SI Query: 1 I/O (rule page – rule covered query).

Example 6.4

This is similar to a previous example, but the column being selected is not in the rule or indexed. This example results in the SI query requiring marginally more I/O, hence it is more expensive to execute than the original query. This is because it uses the same query plan as the original query for part of the SI query (which is 2 queries union-ed), and indexed access to the second part of the union-ed query. The total I/O therefore is the sum of a table scan plus the cost of the indexed access.

Original Query:

```
1> select Description from TClnt
2> where profession = '160'
```

Rule: `if Profession = '160' then PrInd = 'Y' (95% confidence)`

SI Query:

```
1> select Description from TClnt
2> where profession = '160'
3> and (PrInd < 'Y' or PrInd > 'Y')
4> union
```

```
5> select Description from TClnt
6> where profession = '160'
7> and PrInd = 'Y'
```

Original Query: 6154 I/Os
SI Query: 6385 I/Os

Example 6.5:

This reduces I/O by two-thirds of the original query, from 33957 to 11745, by using indexed access instead of a table scan. However, the reduction is not as great as some of the previous examples (examples 6.1 and 6.2) because the SI query is a union'ed query, requiring access for each of the 2 parts of the query.

Original Query:

```
1> select count(*) from tPortfolio
2> where GroupSector = 13770
```

Rule: `if GroupSector = 13770 then IndexCategory = 8 (99% confidence)`

SI Query:

The SI query is produced from using the rule:

```
if GroupSector = 13770 then IndexCategory = 8 (99% confidence)
```

```
1> select count(*) from tPortfolio
2> where GroupSector = 13770
3> and IndexCategory = 8
4> union
```

```
5> select count(*) from tPortfolio
6> where GroupSector = 13770
7> and (IndexCategory < 8 or IndexCategory > 8)
```

Original Query: 33957 I/Os
SI Query: 11745 I/Os

Example 6.6:

This example enables the SI query to use a useful index and has a rule with 100% confidence. The rule does not cover or answer the query, hence table and index access is required to answer the query.

Original Query:

```
1> select * from TCompRet
2> where CompositeType = 40
```

Rule: `if CompositeType = 40 then ClassificationCode = 157 (100% confidence)`

SI Query:

```
1> select * from TCompRet
2> where CompositeType = 40
3> and ClassificationCode = 157
```

Original Query: 11470 I/Os
SI Query: 11 I/Os

Example 6.7:

This example reduces I/O by changing the access path from a table scan to an indexed path. The reduction in I/O is profound, but not as much as some queries, because the SI query is a union'ed query, requiring access for each of the 2 parts of the query.

Original Query:

```
1> select * from TCompRet where CompositeType = 39
```

Rule: `if CompositeType = 39 then ClassificationCode = 147 (95% confidence)`

SI Query:

```
1> select * from TCompRet where CompositeType = 39
2> and ClassificationCode = 147
3> union
4> select * from TCompRet where CompositeType = 39
5> and (ClassificationCode < 147
6>  or ClassificationCode > 147)
```

Original Query: 11470 I/Os
SI Query: 1936 I/Os

Example 6.8:

This is an 'inverse' example, where the 2 predicates in the where clause conflict with a 100% rule, hence no results will be returned. This is an example of incoherence detection.
This example is different in that it uses the SI algorithm to answer the query indirectly by telling us that it has no result set. This is because there is a 100% confidence rule that the query's predicate conflicts with.

Original Query:

```
1> select * from TClnt where Charity = 'Y' and Deal =
'Y'
```

Rule: `if Charity = 'Y' then Deal = 'N' (100% confidence)`
With SI, a read of the rule page is required where it is determined that the rule:

```
If Charity = 'Y' then Deal = 'N'(100% confidence)
```

'inversely' answers the query (inverse rule covering).

Original Query: 6154 I/O
SI Query: 1 I/O – for the rule page

Example 6.9:

In this example, I/O is reduced in the SI query even though the same access path is used (indexed access). In this case, SI enables greater selectivity of the index, reducing the I/O required by about 85%.

Original Query:

```
1> select distinct Security from TQuote
2> where QuoteCode = 'SETTLEMENT'
```

Rule: `if QuoteCode = 'SETTLEMENT' then MarketCode = 'MM'
(72% confidence)`

SI Query:

```
1> select distinct Security from TQuote
2> where QuoteCode = 'SETTLEMENT'
3> and (MarketCode > 'MM' or MarketCode < 'MM')
4> union
5> select distinct Security from Tquote
6> where QuoteCode = 'SETTLEMENT'
7> and MarketCode = 'MM'
```

Original Query: 58134 I/Os

SI Query: 8724 I/Os

6.6 The Query Examples – Second Data Set

The tables and indexes used for the second data set examples are listed. Following the table name is the list of columns, their datatype and the indexes on the table.

Table: *Login_info*
(login_name char(8)
machine_name char(8)
machine_user_name char(12)
location char(12)
login_time datetime
attempts int)
Indexes:
ix2_login_info (login_name, machine_name)
Number of records: 10000

Table: *Flow*
(flow_no int
flow_type_code char(8)
instrument char(1)
flow_calc_code char(8)
flow_ind char(1)
flow_date dateetime
ccy char(3)
flow_amount float)

Indexes:
ix1_flow (flow_calc_code)
Number of records: 8000000

Table: *Curve*
(curve_id int
curve_type_code char(10)
no_of_instances int
currency_code char(3)
schedule int)
Indexes:
ix1_curve (curve_id) – Primary key
Number of records: 6000000

Table: *Trade*
(trade_no int
instrument char(1)
trade_type char(1)
trade_info_code char(4)
trade_status_code char(8)
trade_date datetime
spot_date datetime
far_date datetime
process_org_id int
subject_org_id int
consideration float
reversed char(1))
Indexes:
ix1_trade (trade_no) – Primary key
ix2_trade (trade_info_code, trade_status_code)
ix3_trade (subject_org_id, process_org_id)
Number of records: 1000000

Table: *Sec*
(Sec_no int
Sec_code char(3)
Sec_type char(5)
Class_name char(15)
Description char(30)
Sec_def_code char(10)
Industry char(10))
Indexes:
ix_sec (sec_no) – Primary key
ix1_sec (sec_def_code, class_name)
Number of records: 5000000

Table: *auth_status*
(auth_category char(3)
auth_type_code char(10)
data_group_code char(15)

description char(255)
auth int
rejected int)
Indexes:
ix1_auth_status (auth_category) – Primary key
ix2_auth_status (data_group_code)
Number of records: 10000000

Example 6.10:

This is a sample query that would be executed by a Security/Audit group to check the machines that logins are from. The same indexed access path is used but I/O is reduced in the SI query because it enables greater selectivity by using 2 indexed columns rather than 1.

Original Query:

```
1> select distinct machine_user_name from login_info
2> where login_name = 'PWalds'
```

Rule: `if login_name = 'PWalds' then machine_name = 'RD-02727' (85% confidence)`

SI Query:

```
1> select distinct machine_user_name from login_info
2> where login_name = 'PWalds'
3> and machine_name = 'RD-02727'
4> union
5> select distinct machine_user_name from login_info
6> where login_name = 'PWalds'
8> and (machine_name < 'RD-02727'
9> OR machine_name > 'RD-02727')
```

Original Query: 2078 I/Os

SI Query: 1426 I/Os

Example 6.11:

This query is similar to the previous one, but the column in the select list is indexed (whereas in the previous query it is not). Hence this is an 'index covered' query, and the improvement can be compared to the non-index covered query above.
Since the rule's consequent is the partial result set it is concatenated to the result set of the SI query.

Original Query:

```
1> select distinct machine_name from login_info
2> where login_name = 'PWalds'
```

Rule: `if login_name = 'PWalds' then machine name = 'RD-02727' (85% confidence)`

SI Query:

```
1> select distinct machine_name from login_info
2> where login_name = 'PWalds'
3> and (machine_name < 'RD-02727'
4> or machine_name > 'RD-02727')
5> union
6> select 'RD-02727'
```

Using Sybase, a SELECT statement is permitted without a FROM clause, for literal values and variables.

Original Query: 1312 I/Os

SI Query: 197 I/Os

Example 6.12:

This query is selecting the type of cash flow from a table storing all types of flows. It retrieves the type of calculation that is used for interest-rate based cash flows.

The data access path and query plan are changed significantly, impacting on I/O. The main change is from a full table scan to using an appropriate index.
Since the rule's consequent is the partial result set it is concatenated to the result set of the SI query.

Original Query:

```
1> select distinct flow_calc_code from flow
2> where flow_type_code = 'INTEREST'
```

Rule: `if flow_type_code = 'INTEREST' then flow_calc_code = 'SIMPLEINT' (99% confidence)`

SI Query:

```
1> select distinct flow_calc_code from flow
2> where flow_type_code = 'INTEREST'
3> and (flow_calc_code < 'SIMPLEINT'
4> or flow_calc_code > 'SIMPLEINT')
5> union
6> select 'SIMPLEINT'
```

Original Query: 198161 I/Os
SI Query: 2097 I/Os

Example 6.13:

This is a ‘rule covered’ query – the rule used by the SI algorithm answers the query completely. With SI, a read of the rule page is required where it is determined that the rule answers the query.

Original Query:

```
1> select distinct curve_type_code from curve
2> where currency_code = 'CZK'
```

Rule: `if currency_code = 'CZK' then curve_type_code = 'INTEREST' (100% confidence)`

Original Query: 479 I/Os
SI Query: 1 I/O (rule covered - assuming a single rule page)

Example 6.14:

This is similar to the previous query but is not rule covered – as the rule is not with 100% confidence.

In this example, SI is actually detrimental to performance, because the table involved has to be accessed more than once, although the same access path is used.

Original Query:

```
1> select count(*) from curve
2> where currency_code = 'USD'
```

Rule: `if currency_code = 'USD' then curve_type_code = 'FX' (79% confidence)`

SI Query:

```
1> select count(*) from curve
2> where currency_code = 'USD'
3> and curve_type_code = 'FX'
4> union
5> select count(*) from curve
6> where currency_code = 'USD'
7> and (curve_type_code < 'FX'
   or curve_type_code > 'FX')
```

Original Query: 394 I/Os
SI Query: 797 I/Os

Example 6.15:

This query is based on trades that have matured (expired or settled in the past).

The rule shows that 90% of the matured trades are foreign exchange (FX) trades, which is because they mature quicker than other types of trades.

SI is shown to provide a huge advantage by reducing I/O significantly. This is due to being able to use an indexed access path instead of a table scan.

Original Query:

```
1> select count(*) from trade
2> where trade_status_code = 'MATURED'
```

Rule: `if trade_status_code = 'MATURED' then trade_info_code = 'FX' (90% confidence)`

SI Query:

```
1> select count(*) from trade
2> where trade_status_code = 'MATURED'
3> and trade_info_code = 'FX'
4> union
5> select count(*) from trade
6> where trade_status_code = 'MATURED'
7> and (trade_info_code < 'FX'
8> or trade_info_code > 'FX')
```

Original Query: 106799 I/Os
SI Query: 2328 I/Os

Example 6.16:

This query is based on looking at the classifications in a security table.

The use of SI reduces I/O significantly. This is due to being able to use an index that covers the query, instead of a table scan.

Original Query:

```
1> select count(*) from sec where class_name = 'ISwapLeg'
```

Rule: `if class_name = 'ISwapLeg' then sec_def_code = 'SPECIFIC' (93% confidence)`

SI Query:

```
1> select count(*)
2> from sec where class_name = 'ISwapLeg'
```

```
3> and sec_def_code = 'SPECIFIC'
4> union
5> select count(*)
6> from sec where class_name = 'ISwapLeg'
7> and (sec_def_code < 'SPECIFIC'
8> or sec_def_code > 'SPECIFIC')
```

Original Query: 8820 I/Os
SI Query: 683 I/Os

Example 6.17:

This query is based on a security table.

The use of SI reduces I/O significantly. This is due to being able to use an index that covers the query, instead of a table scan.

Original Query:

```
1> select distinct source from sec
2> where class_name = 'ISwapLeg'
```

Rule: `if class_name = 'ISwapLeg' then sec_def_code = 'SPECIFIC' (93% confidence)`

SI Query:

```
1> select distinct source from sec
2> where class_name = 'ISwapLeg'
3> and sec_def_code = 'SPECIFIC'
4> union
```

```
5> select distinct source from sec
6> where class_name = 'ISwapLeg'
7> and (sec_def_code < 'SPECIFIC'
8> or sec_def_code > 'SPECIFIC')
```

Original Query: 121441 I/Os

SI Query: 9808 I/Os

Example 6.18:

This query is based on finding out about authorisation groups (for traders that can authorise a trade execution).

I/O is reduced by about 8 times by using the SI query. This is due to being able to use an indexed access path instead of a table scan.

Original Query:

```
1> select distinct data_group_code
2> from auth_status
3> where auth_type_code = 'NEW'
```

Rule:`if auth_type_code = 'NEW' then data_group_code = 'trade_stlmt' (80% confidence)`

SI Query:

```
1> select distinct data_group_code
2> from auth_status where auth_type_code = 'NEW'
3> and (data_group_code < 'trade_stlmt'
4> or data_group_code > 'trade_stlmt')
```

Original Query: 893681 I/Os

SI Query: 114758 I/Os

Example 6.19:

In this example, SI enables a change to the access path, giving some improvement in the computational efficiency of execution. Although the applicable rule has 100% confidence, this is not a rule-covered query.

Original Query:

```
1> select spot_date from trade
2> where process_org_id = 3
```

Rule: `if process_org_id = 3 then subject_org_id = 1 (100% confidence)`

SI Query:

```
1> select spot_date from trade
2> where process_org_id = 3
3> and subject_org_id = 1
```

Original query: 78373 I/Os
SI query: 49316 I/Os

6.7 Results Analysis

Table 6.2 below lists the difference in the I/O required between queries executed in their original state and their corresponding SI query. The last column gives the reason for the difference in I/O – the reason why the SI query uses less I/O or more I/O than the original query.

From the empirical examples, the impact of SI on query processing is most significant for 'rule covered' queries. In this situation, SI can reduce the I/O required from as much as several million I/Os to just 1 single I/O (being that of the rule page). Examples 6.3 and 6.8 from the first dataset, and example 6.13 from the second dataset attest to this.

Example	**Original Query (I/Os)**	**SI Query (I/Os)**	**Reason for Difference**
6.1	6154	364	Enable index access
6.2	6348	37	Enable index covering
6.3	6154	1	Rule covering
6.4	6154	6385	Accessed twice (once via index)
6.5	33957	11745	Enable index access
6.6	11470	11	Rule + index access
6.7	11470	1936	Enable Index access
6.8	6154	1	Incoherence detection (via rule)
6.9	58134	8724	Improved predicate selectivity
6.10	2078	1426	Improved predicate selectivity
6.11	1312	197	Index covering
6.12	198161	2097	Enable index access
6.13	479	1	Rule covering
6.14	394	797	Accessed twice
6.15	106799	2328	Index covering
6.16	8820	683	Enable index access
6.17	121441	9808	Enable index access
6.18	893681	114758	Enable index access
6.19	78373	49316	Enable index access

Table 6.2 – I/O Values Between Original and SI Queries

With rules that have less than 100% confidence, SI proved to be most useful under query conditions where it enabled the use of an index that would not have been used otherwise, because the optimiser would not have deemed it selective enough without the additional clauses or predicates added by the application of the SI algorithm. If by adding a first or an additional indexed column to the predicate *where* clause of the query, so that the optimiser finds it possible and efficient to change the query plan to the one used for the original query, then it was seen to be beneficial for reducing I/O. This was more pronounced where it replaced a table scan. From our examples, it can be seen that where this was the case, I/O was reduced by up to about 20 times – to only 5% of the I/O required by the original query. This is seen in example 6.1 of the first dataset. Examples 6.12, 6.18 and 6.19 of the second dataset also benefited from this type of query optimisation plan transformation. Example 6.10 introduces an extra predicate such that the predicate's column is part of the composite index that is used by the original query. Hence SI improves selectivity because the SI query can use both the columns of the composite index. This narrows the searching.

Where the application of SI enabled index covering, then the impact was even greater. The I/O was reduced by 200 times in example 6.2 of the first dataset. Examples 6.11 and 6.15 of the second dataset also demonstrated the advantage of replacing full table scan by index covered access.

However, if SI leads to a situation where one part of the SI query uses the same plan as the original query, and another part of a union-ed SI query uses a different access plan, as in example 6.4 of the first dataset and example 6.14 of the second dataset, this can lead to more I/O being required. Hence it can be more inefficient in this type of situation, requiring the cost of a full table scan plus the cost of indexed access.

6.8 Conclusion

This chapter has put SI to practical use – by applying it to two large, real world databases. This provides empirical evidence of its usefulness.

A set of queries for each of the databases has been processed using the SI algorithm and the cost of processing the original query was compared to the cost of processing the corresponding SI query.

Chapter 6 has demonstrated, with real-world databases, the conditions where SI is useful, and the situations in which its application is beneficial to varying degrees. Additionally, the conditions where it is not advisable to employ SI were demonstrated. The results are discussed and summarised in Section 6.7.

Chapter 7

Semantic Inequivalence with Synthetic Data Distribution

7.1 Introduction

This chapter focuses on the effect of using SI with a synthetic data distribution, based on the normal distribution. This is undertaken to provide an independent and well established data distribution pattern to evaluate the usefulness of SI.

First of all, an overview of the normal distribution is provided and an explanation as to why it is considered useful for statistical analysis.

Following the overview, a set of example queries are executed against the synthetic data where the query predicate's variable is on varying parts of the normal distribution curve. This enables us to look at changes in impact and effectiveness of the SI algorithm along the distribution. Initially, the query variable is on the 'low end' of the normal distribution. Identical queries are subsequently performed where the query predicate's variable is on the 'high end' of the normal distribution, and at various points in between the two extremes. In all of the example cases the original query's I/O cost is compared with the corresponding SI query's I/O cost.

The queries are devised to exemplify the cases of the SI transformation categories: where predicate selectivity is increased enabling more efficient index access / usage (such as examples 7.1, 7.2, 7.3, 7.4), the situation where the SI query involves 2 different access strategies - full table scan and index based access in place of just a table scan (such as examples 7.5, 7.6, 7.7, 7.8) and where

the association rule answers part of the query and index usage is enabled (such as examples 7.9, 7.10, 7.11). The second category increases the cost of query processing. The other two categories reduce the cost of query processing.

7.2 Normal Distribution

The *normal distribution* is an important statistical distribution. All normal distributions are symmetric and have bell-shaped density curves with a single peak.

To speak specifically of any normal distribution, two quantities have to be specified: the *mean* and the *standard deviation*. The *mean* is where the peak of the density occurs, and the *standard deviation* indicates the spread or girth of the bell curve.

A prominent reason that the normal distribution is considered important is because many psychological and educational variables have an approximate normal distribution. Measures of reading ability, introversion, job satisfaction, and memory are among the many psychological variables approximately normally distributed [29]. Although the distributions are only approximately normal, they are usually quite close. A second reason the normal distribution is considered to be so important is that it is easy for mathematical statisticians to work with. This means that many kinds of statistical tests can be derived for normal distributions. Generally, these tests work very well even if the distribution is only approximately normally distributed [29]. The normal distribution curve is illustrated in Figure 7.1.

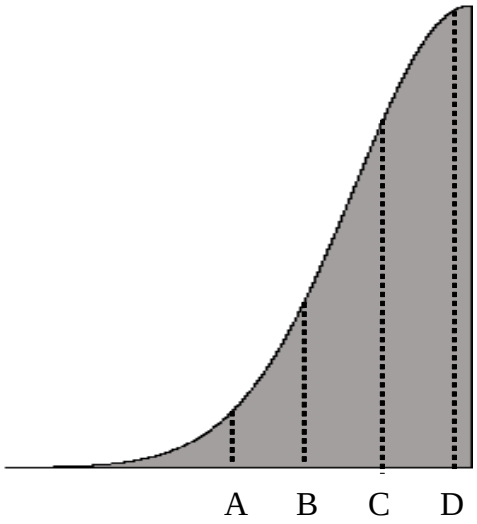

Figure 7.1 – Normal Distribution

In our query examples, the horizontal axis is the attribute value of the query's predicate, such as *subject_type*. The vertical axis represents the frequency of its occurrence – or the number of rows containing that value.

Four positions in the normally distributed data are used for each query example: point A, at the lowest end hence the predicate having a low frequency to point D at the highest end hence the predicate having a high frequency, and at two points in between.

7.3 Query Examples

The query examples for demonstrating the effects of SI with the normal distribution are based on the *titles* table from the example data model set up in Section 3.8.

For the data and queries, a *titles* table was created with the structure defined in the data model, and the column *subject_type* is used as the predicate variable's attribute.

Hence the *subject_type* column is populated with a normally distributed set of values. This single variable is used to exemplify the SI impact along the normal distribution curve.

Table 7.1 shows the 4 data distributions used for the queries in examples 7.1 to 7.4. The points (A, B, C or D) indicate the position of the predicate on the curve in Figure 7.1.

Subject Type	**Predicate variable at low end of normal distribution (point A)**	**Predicate variable at low-mid end of normal distribution (point B)**	**Predicate variable at high-mid end of normal distribution (point C)**	**Predicate variable at high end of normal distribution (point D)**
Astronomy	60	2500	10000	150000
Media	60	60	60	60
Astrology	120	120	120	120
Health	120	120	120	120
Design	250	250	250	250
Travelling	250	250	250	250
Geography	500	500	500	500
Sociology	500	500	500	500
Chemicals	1000	1000	1000	1000
Gardening	1000	1000	1000	1000
Business	2500	2500	2500	2500
Economics	2500	5000	2500	2500
Beauty	5000	5000	5000	5000
History	5000	10000	5000	5000
Biology	10000	10000	10000	10000
Plants	10000	50000	50000	10000
Languages	50000	50000	50000	50000
Science	50000	100000	100000	50000
Maths	100000	100000	100000	100000
Music	100000	150000	150000	100000
Art	150000	60	60	60

Table 7.1 – Data Distributions – Set 1

Initially the data sample is such that the predicate variable, *subject_type* is at the low end of the normal distribution, at point A in Figure 7.1. For subsequent examples, the data distribution is changed, being defined each time, so that the antecedent moves to the high end of the normal distribution, up to point D and at the various points in between. Therefore the differences in the effect of applying SI at various points along the curve in Figure 7.1 can be demonstrated.

The query examples 7.1 to 7.4 are based on the queries and pseudo rule from the example data model defined in Section 3.8:

```
if subject_type = 'Astronomy' then price = 29.95 (70% confidence)
```

The original query used for examples 7.1 to 7.4 is:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
```

The corresponding SI query only asks for the information requested by the original query and unknown from this rule:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
3> and (price < 29.95 or price > 29.95)
```

The subsequent examples (from 7.5 onwards) are based on the rule:

```
if title = 'Maths for beginners' then price = 15.00 (70% confidence).
```

Two groups of queries are used to demonstrate the impact of SI where costs are reduced and where costs are increased.

The detailed query optimiser's processing plan that is output for each query is listed in Appendix A4.

Example 7.1:

With data distribution such that the predicate variable is at the lowest end of the normal distribution curve, as shown in Table 7.1 (point A in Figure 7.1).

Original Query: 79 1/Os
SI Query: 37 I/Os

The SI query is more efficient than the original query by 42 fewer I/Os (79 I/Os for the original and 37 I/Os for the SI query). This is 46% of the original query's I/O – over a 50% improvement. This is due to increased data selectivity enhancing the use of the index.

Example 7.2:

With data distribution changed so that predicate variable is at the lower-mid end of the normal distribution curve, as shown in Table 7.1 (point B in Figure 7.1).

Here the data is changed so that the antecedent is at neither the top end nor the bottom end of the normal distribution – but at the lower-mid end, as can be seen, where `subject_type = 'Astronomy'.`

Original Query: 2666 1/Os
SI Query: 886 I/Os

Here, the benefit is also profound – I/O for the SI query is reduced to just over 33% of that of the original query – from 2666 I/Os to 886 I/Os due to the increased data selectivity reducing the number of pages that need to be accessed.

Example 7.3:

With the data distribution changed again so that the predicate variable is at the middle-upper range on the normal distribution curve, as shown in Table 7.1 (point C in Figure 7.1)

Original Query: 10616 1/Os
SI Query: 3490 I/Os

Here I/O is reduced to less than 33% of the original query from 10616 I/Os to 3490 I/Os. Again this is due to increased data selectivity enhancing the use of the index.

Example 7.4:

With data distribution changed so that predicate variable is at the high or top end of the normal distribution curve, as shown in Table 7.1 (point D in Figure 7.1).

When the data is changed so that `subject_type = 'Astronomy'` is at the top end of the normal distribution, as illustrated in the last column of Table 7.1.

Original Query: 159229 1/Os
SI Query: 33146 I/Os

In this case, with the predicate variable at the high end of the normal distribution, the I/O is reduced by 126083 I/Os - from 159229 I/Os to 33146 I/Os. This is 20% of the I/O of the original query.

The reason for this is because the predicate variable, being at the top end of the normal distribution curve, has had its selectivity increased sufficiently to have made a difference. However, when the variable was at the low end of the normal distribution as in example 7.1, selectivity was relatively high to start with, therefore adding the additional SI predicate did not increase selectivity by the same magnitude.

From the previous 4 examples, we can see that the higher on the normal distribution curve the variable is positioned, the greater the benefit of SI in reducing I/O by a higher proportion.

The rest of the examples are based on the rule:

```
if title = 'Maths for beginners' then price = 15.00
(70% confidence).
```

For each data distribution, there are 2 different queries based on it, for which the same rule is applicable, by using the SI procedure.

The original query used for examples 7.5 to 7.8 is:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
```

The corresponding SI query only asks for the information requested by the original query and unknown from this rule:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
5> select distinct total_sold from titles
```

```
6> where title = 'Maths for beginners'
7> and price = 15
```

The original query used for examples 7.9 to 7.12 is:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
```

The corresponding SI query only asks for the information requested by the original query and unknown from this rule. The rule indicates the existence of *price = 15* for this *title*, hence this is eliminated from the query.

SI Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
5> select 15
```

As noted in example 6.11, using Sybase, a SELECT statement may retrieve a literal value without a FROM clause.

The rule's antecedent is initially at the low end of the normal distribution. Then the data distribution is changed so that it is at intermediate positions on the normal distribution curve and lastly on the highest end with examples included at each point.
The data distributions used for the following queries are detailed in Table 7.2 below. The points (A, B, C or D) indicate the position of the predicate on the curve in Figure 7.1.

Title	Predicate variable at low end of normal distribution (point A)	Predicate variable at low-mid end of normal distribution (point B)	Predicate variable at high-mid end of normal distribution (point C)	Predicate variable at high end of normal distribution (point D)
Maths for beginners	60	2000	20000	100000
World Discovery	60	60	60	60
European Cities	125	125	125	125
Houses and Gardens	125	125	125	125
Cats and Dogs	250	250	250	250
Zoo Animals	250	250	250	250
House Plants	500	500	500	500
Make Up Colour	500	500	500	500
Australia	1000	1000	1000	1000
PC World	1000	1000	1000	1000
Running	2000	60	2000	2000
Yoga for All	2000	2000	2000	2000
Internet Design	3500	3500	3500	3500
Starting on the Internet	3500	3500	3500	3500
Holistic Health	6000	6000	6000	6000
Operating Systems	6000	6000	6000	6000
Java Beans	10000	10000	10000	10000
Networks	10000	10000	10000	10000
Horticulture	20000	20000	100000	20000
Jewellery Design	20000	20000	20000	20000
Gardening	25000	25000	25000	25000
Refloxology for Hands	25000	25000	25000	25000
Algebra	50000	50000	50000	50000
Style	50000	50000	50000	50000
Advanced Maths	75000	75000	75000	75000
Basic Grammar	75000	75000	75000	75000
Costumes	100000	100000	60	60

Table 7.2 – Data Distributions – Set 2

Example 7.5:

With data distribution such that the antecedent is at the low end of the normal distribution curve, as defined in Table 7.2 (point A in Figure 7.1).

Original Query: 77 1/Os
SI Query: 84 I/Os

In this example, there is no advantage in using SI. I/O is increased by 9%. Predicate selectivity was high in the original query, and the SI query did not add sufficient extra selectivity that could reduce I/O. Also the columns selected were not included in the index hence access to underlying data pages was necessary.

Example 7.6:

When the data distribution changed so that the antecedent is on the lower-mid range of the normal distribution, as shown in Table 7.2 (point B in Figure 7.1).

Original Query: 2751 1/Os
SI Query: 2861 I/Os

In this example, with the antecedent at the lower-mid range of the normal distribution, I/O is increased from 2751 to 2861. This is a 4% increase. Again, this is because predicate selectivity was high in the original query, and the SI query did not add sufficient extra selectivity that could reduce I/O.

Example 7.7:

When the antecedent is changed so that it is at the mid-higher end of the normal distribution curve (point C in Figure 7.1).

Original Query: 41534 1/Os
SI Query: 62823 I/Os

This example shows that SI has actually increased the I/O from 41534 to 62823, - nearly a 50% increase. Since indexed access is not used for this query, an extra table scan is required, causing the large increase in I/O.

Example 7.8:

With data distribution such that antecedent is at the highest end of the normal distribution curve (point D in Figure 7.1).

Original Query: 107837 1/Os
SI Query: 112476 I/Os

This gives no improvement in I/O, but actually increases the cost from 107837 to 112476 I/Os. This is a 4% increase. Selectivity is not increased enough to effect the query plan, since predicate selectivity was high in the original query.

Example 7.9:

This example query is based on the same data distribution as example 7.5.

Original Query: 75 1/Os
SI Query: 33 I/Os

With the antecedent at the low end, the SI query has reduced I/O to 44% of the original query – over 50% improvement, from 75 to 33 I/Os. The indexed access covers the query and selectivity is increased enough to reduce the I/O required.

Example 7.10:

This example query is based on the same data distribution as example 7.6.

Original Query: 2130 1/Os

SI Query: 909 I/Os

SI has reduced I/O to 42% of that required by the original query, from 2130 to 909. Again, this is because the indexed access covers the query and selectivity is increased enough to reduce the I/O required.

Example 7.11:

This example query is based on the same data distribution as example 7.7.

Original Query: 21077 1/Os

SI Query: 6825 I/Os

In this example, I/O required by the SI query has been reduced to 33% of the I/Os of the original query from 21077 to 6825 for the same reasons as the previous example.

Example 7.12

This example query is based on the same data distribution as example 7.8.

Original Query: 104079 1/Os

SI Query: 32777 I/Os

This gives a large improvement in I/O - from 104079 to 32777, which is only 31% of the I/Os of the original query, due to increased selectivity enabling increase in the use of the index.

7.4 Conclusion

For the sample query set, we can see that SI is generally better where the antecedent is at the high end of the normal distribution. This is because the application of the SI algorithm increases the selectivity by a greater order of magnitude compared to when the antecedent is at the low end.

Table 7.3 shows the rounded average improvement in I/O in relation to where the antecedent lies on the normal distribution of the queries, where an improvement is noted.

Position on normal distribution	**Approximate Average % improvement**
Highest end (point D)	75%
Upper middle (point C)	70%
Lower middle (point B)	60%
Lowest end (point A)	50%

Table 7.3 – Average I/O Improvement with SI and Normal Distribution

The following graph similarly represents the effect of SI in terms of the proportion of I/O used for the transformed query depending on where on the normal distribution curve the antecedent is. This takes all the sample queries into account including where there is no improvement due to there being no change in the data access path.

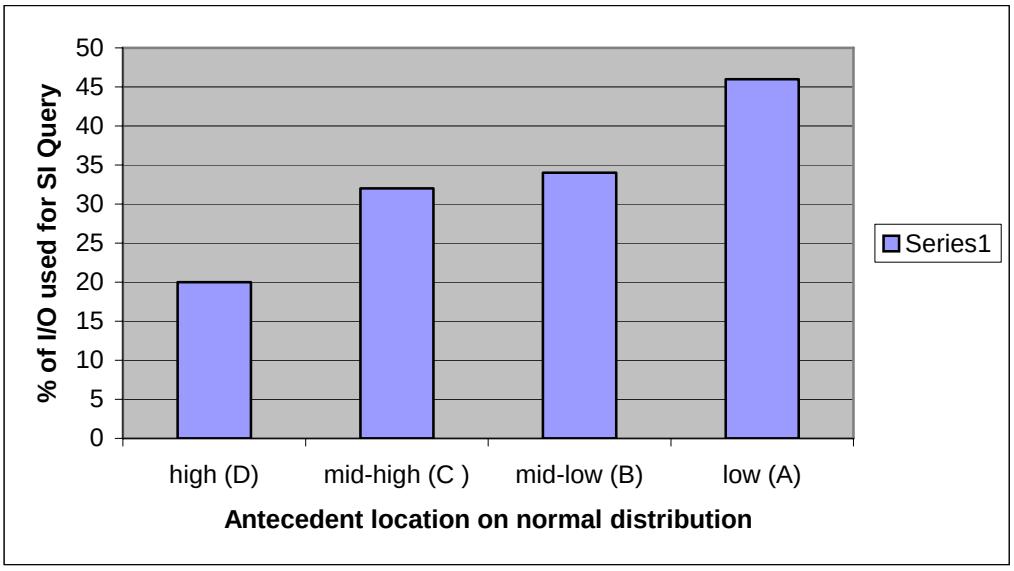

Figure 7.2 – Improvement for Normally Distributed Data with SI

Improvement can also be seen where the antecedent is at point A, the low end of the normal distribution, although it is not by the same proportion because selectivity is relatively high to start with in such cases. Moreover, the SI improvement is greater where the increased selectivity is supported by an indexed access strategy.

From the examples, it can also be seen that SI provides a distinct advantage where the rule actually reduces the query being asked – that is, it especially helps where less data is requested because the rule answers some of the original query. This is more profound the higher up the normal distribution curve that the antecedent is, although the advantage in this situation can be seen all the way through, but to a lesser degree on the lower end of the normal distribution curve.

Chapter 8

Concluding Remarks and Further Research

8.1 Introduction

The work presented in this thesis has been motivated by the scope for exploitation in query optimisation of the potentially valuable information that association rules, produced from database mining, can provide.

This thesis aims to help fill this gap by introducing a new strategy in query processing that brings together two areas of DBMS research. These are data mining of association rules and query optimisation. How they can be used together in a complementary and novel way is analysed by proposing the new concept of SI.

This concluding chapter presents a summary of what the thesis has discussed and achieved. Additionally, it identifies some potential areas for further related research.

8.2 Research Summary

Following a review of related work on data mining of association rules and DBMS query optimisation, the new concept of SI was introduced and discussed in detail. How it is distinguishable from other query processing strategies and can

add value in situations, where the current or existing strategies cannot, was also examined.

For SI to be practical, it should be able to be used within an existing extensible database query optimiser. For it to be usable as a new strategy within an existing DBMS query optimiser architecture, it is defined in a modular, encapsulated fashion. Hence it is shown how it can be incorporated into the extensible architecture as a new query processing module, or component.

The thesis has defined an algorithm for implementing SI, and representative database queries processed by stepping through the algorithm. The query I/O costs are compared before using SI with the cost of the post-SI transformed query.

Following this, empirical evidence of the value of SI has been demonstrated by using queries against two large real-world databases and comparing the respective costs of processing both the original and the corresponding SI query. This was followed by a discussion on when SI is most appropriate to use. For this, a wide range of queries were used to help identify the situations where SI resulted in increased efficiency. The types of situations that can give rise to varying degrees of improvement were reviewed. Moreover, the situations that do not benefit from the application of SI were also identified. Exceptional cases and the effect that SI has on them were also analysed and presented.

To reduce the potential of bias in a single dataset, two independent real-world databases were used for producing empirical results. In addition, a synthetic data distribution based on the normal distribution was used to test SI against.

For facilitating further analysis, the independent well-established Decomposition Algorithm for query processing was introduced. This enabled the comparison of the costs of using SI with the costs of not using it, under a completely separate query processing costing algorithm.

8.3 Review of Aims and Accomplishments

The aim of using the output from the data mining of association rules in query optimisation is achieved by SI providing a link between the two areas. The reason for the introduction of SI is to reduce the query processing costs in some situations where existing methods fail to achieve the same cost reduction. By using the database association rules, query processing is extended beyond using column value distribution statistics to being able to use relationships that exist between data values held in the database. This increases the input to the query optimisation process by providing more information than was previously made available to it. The information may be sourced from the vast research carried out in efficient database association rule mining or from any other source that produces similar output. The use of the extra information that association rules provide is demonstrated in both the situations where it can and the situations where it cannot reduce the cost of answering the query.

Generally, SI reduces the cost of processing a query in situations where the data access path used by the DBMS is more efficient for the SI query than for the original query. The biggest advantage was seen where a rule can ‘cover’ or completely answer a query, or demonstrates that there is a null or empty result set.

Relatively large improvements were seen when the SI query is able to use an index where previously a full table scan was required. The SI transformation process was also useful where the data access path was the same but the selectivity was increased so that the indexed search, for example, was narrowed earlier in the index traversal procedure.

However, where the SI query resulted in the same access path, or resulted in a union query with each part requiring a different access strategy, it was less efficient to answer than the original query.

These results were seen with both the synthetic data and the real-world data.

With the normally distributed data, SI overall improves the efficiency of query processing; however, the improvement was greater where the rule antecedent had a higher frequency (higher up on the curve).

It is seen that in addition to encouraging use of concepts such as rule pages and rule covering, SI can also be used in conjunction with existing researched concepts, such as partial indexing in a complementary manner. This adds to the value of related existing research.

8.4 Further Research

The research started in this thesis can be taken further and built upon in several ways.

SI has been studied in respect of SQL select-project queries, excluding aggregates, group by and having clauses. Investigation of a wider class of queries in particular those involving joins, remains for future work.

The thesis has studied SI with respect to relational databases only. However, SI can be researched with non-relational databases, such as object-oriented or network databases. Its applicability and usefulness can similarly be considered and a SI query processing region or component introduced into an object-oriented (or alternatively structured) database's extensible query optimiser. A limitation to be aware of is that there is not so vast a base of research on data mining of association rules for databases other than relational.

Although SI is considered more suitable for very large databases such as data warehouses, where the skewness of the data is relatively stable, research could be carried out to investigate the effects of varying degrees of changes in the data distribution. The impact of SI on query processing efficiency could be investigated. The point when it becomes feasible or profitable to update the

association rules that are used as input may be studied in more detail, as well as the data change volatility impact on the suitability of using SI.

SI and partial indexing are complementary. Research into using these strategies in conjunction with each other can be taken further. This may look into situations, for example, where a useful high confidence rule is found; the question then arises as to whether the remainder of the data should be partially indexed? What space saving and performance advantage could this achieve? The various trade-offs and turning points in profitability can thus be analysed.

References

1. K Aberer and G Fischer
Semantic Query Optimisation for Methods in Object Oriented Database Systems
IEEE International Conference on Data Engineering, Taipei, March 1995, pp 70–79.

2. R Agrawal and R Srikant
Fast Algorithms for Mining Association Rules in Large Databases
International Conference on Very Large Databases, Santiago, September 1994, pp 487-500.

3. R Agrawal, T Imielinski and A Swami
Mining Association Rules Between Sets of Items in Large Databases
Proceedings of ACM SIGMOD International Conference on Management of Data, Washington, May 1993, pp 207-216.

4. D Beneventano, S Bergamaschi and C Sartori
Description Logics for Semantic Query Optimization in Object-Oriented Database Systems
ACM Transactions on Database Systems, Volume 28, Issue 1, March 2003, pp 1-50.

5. S Ceri and J Widom
Managing Semantic Heterogeneity with Production Rules and Persistent Queries
International Conference on Very Large Databases, Dublin, August 1993, pp 108-119.

6. U Chakravarthy, J Grant and J Minker
Logic-Based Approach to Semantic Query Optimization

ACM Transactions on Database Systems, Volume 15, Issue 2, June 1990, pp162-207

7. C Chan, B Ooi and H Lu
Extensible Buffer Management of Indexes
International Conference on Very Large Databases, Vancouver, August 1992, pp 444-454.

8. R Choenni and A Siebes
A Framework for Query Optimisation to Support Data Mining
Computer Science/Department of Algorithmics and Architecture, Report CS-R9637 ISSN 0169-118X, Proceedings of the International Workshop on Database and Expert Systems Application, Toulouse, September 1997.

9. H Darwen
The Role of Functional Dependence in Query Decomposition
Relational Database Writings 1989-1991, Addison-Wesley 1992, Chapter 10, pp 133-154.

10. D Das
Making Database Optimisers More Extensible
PhD thesis, University of Texas at Austin, 1995.

11. D Das and D Batory
Prairie: An Algebraic Framework for Rule Specification in Query Optimisers
IEEE International Conference on Data Engineering, Taipei, March 1995, pp 201-210.

12. R Elmasri and S Navathe
Fundamentals of Database Systems
Addison-Wesley, 4th edition 2003.

13. U Fayyed, G Piatetsky-Shapiro, R Uthurusamy

Panel and workshop reports from KDD-2003: Data Mining: the next 10 years
ACM SIGKDD Explorations Newsletter, Volume 5, Issue 2, December 2003, pp 191-196.

14. L Feagres, D Maier and T Sheard
Specifying Rule based Query Optimisers in a Reflective Framework
IEEE International Conference on Deductive and Object-Oriented Databases, Phoenix, December 1993, pp 146-168.

15. J Freytag
A Rule Based View of Query Optimisation
Proceedings of ACM SIGMOD International Conference on the Management of Data, San Francisco, May 1987, pp 173-180.

16. P Godfrey, J Gryz and C Zuzarte
Exploiting Constraint-like Data Characterizations in Query Optimisation
Proceedings of ACM SIGMOD International Conference on the Management of Data, California, May 2001, pp 582-592.

17. G Graefe
Volcano: An Extensible and Parallel Query Evaluation System
University of Colorado at Boulder, Technical Report No 481, 1990.

18. L Haas, J Freytag, G Loman and H Pirahesh
Extensible Query Processing in Starburst
Proceedings of ACM SIGMOD International Conference on the Management of Data, Oregon, May 1989, pp 377-388.

19. J Han and Y Fu
Discovery of Multiple Level Association Rules from Large Databases
J Han and Y Fu, International Conference on Very Large Databases, Zurich, September 1995, pp 420-431.

20. J Han, Huang, N Cercone and Y Fu

Intelligent Query Answering by Knowledge Discovery Techniques
IEEE Transactions on Knowledge and Data Engineering, Volume 8, Issue 3, June 1996, pp 373-390.

21. C Hsu and C Knoblock
Rule Induction for Semantic Query Optimization
Proc of the 11th International Conference on Machine Learning, New Brunswick, NJ, 1994, pp 112-120

22. Y Huhtala, J Karkkainen, P Porkka and H Toivonen
Efficient Discovery of Functional and Approximate Dependencies Using Partitions
International Conference on Data Engineering, Orlando, February 1998, pp 392-401.

23. K Joshi
Analysis of Data Mining Algorithms
Added to www.gl.umbc.edu/~kjoshi1/data-mine/proj_rpt.htm in March 2004.

24. C Kilger and G Moerkotte
Indexing Multiple Sets
International Conference on Very Large Databases, Santiago, September 1994, pp 180-191.

25. W Kim, K Kim and A Dale
Indexing Techniques for Object Oriented Databases
Object Oriented Concepts, Databases and Applications, Addison-Wesley, 1989, pp 371-394.

26. W Kim, D Reiner and D Batory (editors)
Query Processing in Database Systems
Springer-Verlag, 1995.

27. J J King

QUIST: A System for Semantic Query Optimization in Relational Databases
International Conference on Very Large Data Bases, Cannes, September 9-11, 1981, pp 510-517

28. M Klemettinen, H Mannila, P Ronkainen, H Toivonen and A Verkamo
Finding Interesting Rules from Large Sets of Discovered Association Rules
IEEE International Conference on Information and Knowledge Management, Maryland, December 1994, pp 401-407.

29. D Lane
HyperStat Online Textbook, http://www.davidmlane.com/hyperstat/
Professor of Psychology, Statistics and Management, Rice University, 2003.

30. H Mannila, H Toivonen and A Verkamo
Efficient Algorithms for Discovering Association Rules
AAAI Workshop of Knowledge Discovery in Databases, Seattle, July 1994, pp 181-192.

31. G Mitchell
Extensible Query Processing in an Object Oriented Database
PhD thesis, Brown University, May 1993

32. Oracle Corporation
Oracle Product Manuals.

33. H Pang, H Lu, and B Ooi.
An Efficient Semantic Query Optimization Algorithm.
IEEE International Conference on Data Engineering, Kobe Japan, April 1991 pp 326-335.

34. J Park, M Chen and P Yu
An Effective Hash - Based Algorithm for Mining Association Rules
ACM SIGMOD International Conference on the Management of Data, San Jose, May 1995, pp 175-186.

35. T Ravindra-Babu, M Narasimha-Murty, V Agrawal
Hybrid Learning Scheme for Data Mining Applications
Fourth International Conference on Hybrid Intelligent Systems, Kitakyushu, Japan, December 2004, pp 266-271.

36. S Sarawagi, S Thomas and R Agrawal
Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications
ACM SIGMOD International Conference on the Management of Data, Seattle, June 1998, pp 343-355.

37. C Sartori and M Scalas
Partial Indexing for Non Uniform Data Distributions in Relational Database Management Systems
IEEE Transactions on Knowledge and Data Engineering, Volume 6, Issue 3, June1994, pp 420-429.

38. A Savasere, E Omiecinski and S Navathe
An Efficient Algorithm for Mining Association Rules in Large Databases
International Conference on Very Large Databases, Zurich, September 1995, pp 432-444.

39. 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 Volume 5, Issue 6, December 1993, pp 950-964.

40. S Shenoy and Z Ozsoyoglu
Design and Implementation of a Semantic Query Optimizer
IEEE Transactions on Knowledge and Data Engineering, Volume 1, Issue 3, September 1989, pp 344-361.

41. M Siegel, E Sciore and S Salveter
A Method for Automatic Rule Derivation to Support Semantic Query Optimisation
ACM Transactions on Database Systems, Volume 17, Issue 4, December 1992, pp 563-600.

42. R Srikant and R Agrawal
Mining Generalised Association Rules
International Conference on Very Large Databases, Zurich, September 1995, pp 407-419.

43. R Srikant and R Agrawal
Mining Quantitative Association Rules in Large Relational Tables
Proceedings of ACM SIGMOD International Conference on Management of Data, Montreal, June 1996, pp 1-12.

44. M Stonebraker, L Rowe and M Hirohama
The Implementation of POSTGRES
IEEE Transactions on Knowledge and Data Engineering Volume 2, Issue 1, March 1990, pp 125-142.

45. Sybase Inc.
Sybase Adaptive Server Enterprise Manuals.

46. T Topaloglou, A Illarramandi, and L Sbattella
Query Optimisation for Knowledge Base Management Systems: Temporal, Syntactic and Semantic Transformations
IEEE International Conference on Data Engineering, Tempe, February 1992, pp 310-319.

47. A Trigoni
Ch 6: Using Association Rules to Optimize Queries Semantically from '*Semantic Optimization of OQL Queries'*

Technical Report No 547, ISSN 1476-2986, October 2002, University of Cambridge Computer Laboratory

48. A Trigoni and K Moody
Using Association Rules to Add or Eliminate Query Constraints Automatically
IEEE Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management (SSDBM 01), 2001, pp 124-133.

49. Whitecross Systems Inc.
Mining Very Large Databases to Support Knowledge Exploration
Whitepaper, Jan 2001.

50. E Wong and K Yousefi
Decomposition - A Strategy for Query Processing
ACM Transactions on Database Systems, Volume 1, Issue 3, New York, September 1976, pp 223-241.

51. C Yu and W Sun
Automatic Knowledge Acquisition and Maintenance for Semantic Query Optimization
IEEE Transactions on Knowledge and Data Engineering, Volume 1, Issue 3, September 1989, pp 362-375.

52. N Zhong, L Chunnian, Y Yao, M Ohshima, M Huang, J Huang
Relational Peculiarity Oriented Data Mining
IEEE International Conference on Data Mining, Brighton, November 2004, pp 575-578.

Appendix A1

In this appendix, some terms and keywords that are used throughout the thesis are defined. The definitions are with respect to relational databases. Terms in *italics* are subsequently defined in the ensuing list of definitions

Approximate dependency

A *functional dependency* that almost holds. Some rows can contain exceptions to the stated dependency. This is an alternative name for an association rule [30].

Bitmap Index

An index that uses a string of bits that corresponds to rows in a table to indicate whether the indexed value is stored in a row. There is a bit string for each possible data value [12].

Clustered Index

An *index* where the data is physically stored in the order of the indexed columns. This contrasts to a non-clustered index where the storage order of data in the table is not related to the indexed keys (or columns) [12].

Confidence

The probability that a row contains both the antecedent and the consequent of a rule given that the antecedent occurs. The *confidence* statistic is the measure of a rule's strength [12].

Database page

A unit of storage for the database objects. Data pages store data rows for the tables, index pages store index nodes for the indexes [45].

Data Mining of Association Rules

A process for discovering association rules from a large database. This is also known as knowledge discovery [12].

Functional dependency

This states that the value of an attribute (or set of attributes) is uniquely determined by the value of some other attribute (or set of attributes) [12].

Index

A database object that can speed access to specific data rows by providing an access path allowing direct access to data based on an index term [12].

Integrity constraints

Enforce the data values that are acceptable for certain attributes [12].

Optimiser extensibility

The ability to add new query processing strategies to the database management system's optimiser [17].

Partial indexes

A partial index is an index that has some condition applied to it such that it only includes a portion of the rows in a table. This can allow the index to remain small even though the table may be rather large, and have fairly extreme selectivity [37].

Query optimisation

The process of analysing a query to find out what resources are needed to answer it and how the resources can be minimized to answer the query more efficiently [12].

Rule covering

This is where a rule has 100% confidence and may be used to completely answer a query if the query's request has all parts satisfied by the rule.

Rule page

This is a new concept introduced in the thesis. A rule page is a database page for storing the association rules that are relevant to the database querying patterns.

Semantic query optimisation

A query optimisation strategy whereby a query is transformed based on the functional dependencies known about the data. It maintains the semantics or meaning of the query [12].

Appendix A2

The following typefaces are used throughout the thesis for the purposes defined.

Times New Roman is used for general text.

`Courier New` is used for code fragments.

Italic is used for definitions, formulae and algorithms. It is also used for synonyms and variables.

Arial Narrow is used for query optimiser output and the Decomposition Algorithm output.

Appendix A3

This lists the main abbreviations used throughout the thesis.

DBMS - Database Management System

DSS - Decision Support System

I/O – Input / Output

LRU - Least Recently Used

MIS - Management Information System

MRU - Most Recently Used

NPAG – Number of Pages

OLTP – On-Line Transaction Processing

RDBMS – Relational Database Management System

SARG – Search Argument

SI - Semantic Inequivalence

SQL - Structured Query Language

SQO – Semantic Query Optimisation

VLDB – Very Large Database

Appendix A4

This appendix contains the detailed query plans produced by the optimiser, of the original queries and corresponding SI queries that are used in Chapters 6 and 7.

A short explanation of the Sybase Adaptive Server query plans is:

Step: output displays "STEP N" for every query, where N is an integer, beginning with "STEP 1". For some queries, Adaptive Server cannot retrieve the results in a single step and breaks the query plan into several steps.
From Table: indicates which table the query is reading from. The "FROM TABLE" message is followed on the next line by the table name.
To Table: for operations that require an intermediate step to insert rows into a worktable, "TO TABLE" indicates that the results are going to the "Worktable" table rather than to a user table.
Nested Iteration: indicates one or more loops through a table to return rows.
Table Scan: indicates the query performs a table scan.
Clustered Index: indicates that the query optimizer chose to use the clustered index on a table to retrieve the rows.
Index Name: indicates that the query is using an index to retrieve the rows. The message includes the index name.
Scan Direction: indicate the direction of a table or index scan – can be Forward scan or Backward scan.
Index Covering: indicates that an index covers the query.
Keys: indicates the indexed columns used when an index is used to locate rows.
I/O Size: this reports the I/O size used in the query.
Cache Strategy: indicates the buffer cache replacement strategy used for data pages and for index leaf pages - least recently used (LRU) pages or most recently used (MRU).

Chapter 6 Queries – First Data Set

Example 6.1:

Original Query: select * from TClnt where Ccy = 'ZAR'

Rule: if Ccy = 'ZAR' then CtryResidenceCode = 220 (90% confidence)

Original Query: 6154 I/Os
SI Query: 364 I/Os

Original Query:

```
1> select * from TClnt
2> where Ccy = 'ZAR'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SELECT.

FROM TABLE
TClnt
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

Table: TClnt scan count 1, logical reads: (regular=6154 apf=0 total=6154),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query**:**

```
1> select * from TClnt where ccy = 'ZAR'
2> and ctryresidencecode = 220
3> union
4> select * from TClnt where ccy = 'ZAR'
5> and (ctryresidencecode < 220
6>    or ctryresidencecode > 220)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

FROM TABLE

TClnt

Nested iteration.

Index : Clnt_x2

Forward scan.

Positioning by key.

Keys are:

CtryResidenceCode ASC

Ccy ASC

Using I/O Size 16 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 2 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 1

The type of query is INSERT.

The update mode is direct.

FROM TABLE

TClnt

```
    Nested iteration.
    Index : Clnt_x2
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
         CtryResidenceCode  ASC
       Ccy  ASC
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
       Worktable1.

  STEP 1
    The type of query is INSERT.
    The update mode is direct.

    FROM TABLE
       TClnt
    Nested iteration.
    Index : Clnt_x2
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
       CtryResidenceCode  ASC
       Ccy  ASC
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

FROM TABLE
       TClnt
    Nested iteration.
    Index : Clnt_x2
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
```

```
        CtryResidenceCode  ASC
        Ccy  ASC
      Using I/O Size 2 Kbytes for index leaf pages.
      With LRU Buffer Replacement Strategy for index leaf pages.

      FROM TABLE
        TClnt
      Nested iteration.
      Using Dynamic Index.
      Forward scan.
      Positioning by Row IDentifier (RID).
      Using I/O Size 2 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.
      TO TABLE
        Worktable1.

   STEP 1
      The type of query is SELECT.
      This step involves sorting.

      FROM TABLE
        Worktable1.
      Using GETSORTED
      Table Scan.
      Forward scan.
      Positioning at start of table.
      Using I/O Size 16 Kbytes for data pages.
      With MRU Buffer Replacement Strategy for data pages.
Table: TClnt scan count 1, logical reads: (regular=7 apf=0 total=7),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable2 is done in Serial

Table: TClnt scan count 2, logical reads: (regular=316 apf=0 total=316),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable2  scan count 1, logical reads: (regular=20 apf=0 total=20),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=21 apf=0 total=21),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
```

Total writes for this command: 5

Example 6.2:

Original Query: select distinct PrInd from TClnt where profession = '160'

Rule: if Profession = '160' then PrInd = 'Y' – 95% confidence

Original Query: 6348 I/Os
SI Query: 37 I/Os

In this example, I/O is reduced by 200 times. This is because the query optimiser decides to full table scan for the original query, while the SI query enables an index to be used which covers the query so that the underlying table does not need to be accessed. The index answers the SI query.

Original Query:

```
1> select distinct PrInd from TClnt
2> where profession = '160'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
TClnt
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: TClnt scan count 1, logical reads: (regular=6154 apf=0 total=6154), physical reads: (regular=0 apf=0 total=0), a
pf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=194 apf=0 total=194), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct PrInd from TClnt
2> where profession = '160'
3> and (PrInd < 'Y' or PrInd > 'Y')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

```
        FROM TABLE
            TClnt
        Nested iteration.
        Index : Clnt_x3
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            PrInd  ASC
            Profession  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.

        FROM TABLE
            TClnt
        Nested iteration.
        Index : Clnt_x3
        Forward scan.
        Positioning by key.
        Index contains all needed columns. Base table will not be read.
        Keys are:
            PrInd  ASC
            Profession  ASC
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.

FROM TABLE
            TClnt
        Nested iteration.
        Using Dynamic Index.
        Forward scan.
        Positioning by Row IDentifier (RID).
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.
```

```
    FROM TABLE
      Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable2 is done in Serial

The sort for Worktable1 is done in Serial

Table: TClnt scan count 2, logical reads: (regular=7 apf=0 total=7), physical reads: (regular=0 apf=0
total=0), apf IOs
 used=0
Table: Worktable1  scan count 0, logical reads: (regular=12 apf=0 total=12), physical reads: (regular=0
apf=0 total=0), apf
IOs used=0
Table: Worktable2  scan count 1, logical reads: (regular=18 apf=0 total=18), physical reads: (regular=0
apf=0 total=0), apf
IOs used=0
Total writes for this command: 5
```

Example 6.3:

Original Query: select distinct Deal from TClnt where Charity = 'Y'

Rule: if Charity = 'Y' then Deal = 'N' (100% confidence)

Original query: 6154 I/Os
SI query: 1 I/O (rule page – rule covered query).

This example demonstrates the most powerful use of SI – where the rule has 100% confidence and ‘covers’ or answers the SI query without the need to access the table or index.

Original Query:

```
1> select distinct Deal from TClnt
2> where Charity = 'Y'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
TClnt
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: TClnt scan count 1, logical reads: (regular=6154 apf=0 total=6154),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=925 apf=0 total=925),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

versus: 1 read of the rule page via rule covering. There are no indexes on either of these columns.

The rule is `(if Charity = 'Y' then Deal = 'N') – 100% confidence`

Example 6.4

Original Query: select Description from TClnt where profession = '160'

Rule: if Profession = '160' then PrInd = 'Y' (95% confidence)

Original Query: 6154 I/Os
SI Query: 6385 I/Os

Original Query:

```
1> select Description from TClnt
2> where profession = '160'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SELECT.

FROM TABLE
TClnt
Nested iteration.
Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

Table: TClnt scan count 1, logical reads: (regular=6154 apf=0 total=6154), physical reads: (regular=21 apf=756 total=777), apf IOs used=756

Total writes for this command: 0

SI Query:

```
1> select Description from TClnt
2> where profession = '160'
3> and (PrInd < 'Y' or PrInd > 'Y')
4> union
5> select Description from TClnt
6> where profession = '160'
7> and PrInd = 'Y'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

FROM TABLE

TClnt

Nested iteration.

Index : Clnt_x3

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

PrInd ASC

Profession ASC

Using I/O Size 2 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

```
FROM TABLE
        TClnt
      Nested iteration.
      Index : Clnt_x3
      Forward scan.
      Positioning by key.
      Index contains all needed columns. Base table will not be read.
      Keys are:
        PrInd  ASC
        Profession  ASC
      Using I/O Size 2 Kbytes for index leaf pages.
      With LRU Buffer Replacement Strategy for index leaf pages.

      FROM TABLE
        TClnt
      Nested iteration.
      Using Dynamic Index.
      Forward scan.
      Positioning by Row IDentifier (RID).
      Using I/O Size 2 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.
      TO TABLE
        Worktable1.

   STEP 1
      The type of query is INSERT.
      The update mode is direct.

      FROM TABLE
        TClnt
      Nested iteration.
      Table Scan.
      Forward scan.
      Positioning at start of table.
      Using I/O Size 16 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.
      TO TABLE
        Worktable1.
STEP 1
      The type of query is SELECT.
```

This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable2 is done in Serial

Table: TCInt scan count 2, logical reads: (regular=7 apf=0 total=7), physical reads: (regular=0 apf=0 total=0), apf IOs
used=0
Table: Worktable2 scan count 1, logical reads: (regular=18 apf=0 total=18), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: TCInt scan count 1, logical reads: (regular=6154 apf=0 total=6154), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1 scan count 0, logical reads: (regular=206 apf=0 total=206), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 6

Example 6.5:

Original Query: select count(*) from tPortfolio where GroupSector = 13770

Rule: if GroupSector = 13770 then IndexCategory = 8 (99% confidence)

Original Query: 33957 I/Os
SI Query: 11745 I/Os

This reduces I/O by two-thirds of the original, by using indexed access rather than a table scan. However, the reduction is not as great as some of the previous examples because the index is accessed twice.

Original Query:

```
1> select count(*) from tPortfolio
2> where GroupSector = 13770
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
      tPortfolio
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

  STEP 2
    The type of query is SELECT.

Table: tPortfolio scan count 1, logical reads: (regular=33957 apf=0
total=33957), physical reads: (regular=8 apf=5026 total=5034), apf IOs
used=5026
Total writes for this command: 0
```

SI Query:

```
1> select count(*) from tPortfolio
2> where GroupSector = 13770 and IndexCategory = 8
```

```
3> union
4> select count(*) from tPortfolio
5> where GroupSector = 13770
6> and (IndexCategory < 8 or IndexCategory >8)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
      tPortfolio
    Nested iteration.
    Index : Pf_ix3
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
      IndexCategory  ASC
      GroupSector  ASC
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

  STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
      Worktable1.
STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
      tPortfolio
    Nested iteration.
    Index : Pf_ix3
    Forward scan.
    Positioning at index start.
```

```
    Index contains all needed columns. Base table will not be read.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

  STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
      Worktable1.

  STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
      Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: tPortfolio scan count 1, logical reads: (regular=4274 apf=0
total=4274), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: tPortfolio scan count 1, logical reads: (regular=7463 apf=0
total=7463), physical reads: (regular=8 apf=380 total=388), apf IOs used=394
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=10 apf=0 total=10),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

Example 6.6:

Original Query: select * from TCompRet where CompositeType = 40

Rule: if CompositeType = 40 then ClassificationCode = 157 (100% confidence)

Original Query: 11470 I/Os
SI Query: 11 I/Os

This example enables the SI query to use a useful index and has a rule with 100% confidence. The rule does not 'cover' or answer the query, hence table and index access are required to answer the query.

Original Query:

```
1> select * from TCompRet
2> where CompositeType = 40
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.

    FROM TABLE
      TCompRet
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: TCompRet scan count 1, logical reads: (regular=11470 apf=0
total=11470), physical reads: (regular=453 apf=930 total=1383), apf IOs
used=930
Total writes for this command: 0
```

SI Query:

```
1> select * from TCompRet
```

```
2> where CompositeType = 40
3> and ClassificationCode = 157
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SELECT.

FROM TABLE
TCompRet
Nested iteration.
Index : ixcompreturn3
Forward scan.
Positioning by key.
Keys are:
ClassificationCode ASC
CompositeType ASC
Using I/O Size 16 Kbytes for index leaf pages.
With MRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

Table: TCompRet scan count 1, logical reads: (regular=11 apf=0 total=11),
physical reads: (regular=7 apf=0 total=7), apf IOs used=0
Total writes for this command: 0

Example 6.7:

Original Query: select * from TCompRet where CompositeType = 39

Rule: If CompositeType = 39 then ClassificationCode = 147 (95%)

Original Query: 11470 I/Os
SI Query: 1936 I/Os

This example reduces I/O by changing the data access path from a table scan to an index. The reduction is I/O is profound, but not as much as some queries because the SI query is a union'ed query – and the index is accessed twice – once for each part of the union-ed query.

Original Query:

```
1> select * from TCompRet
2> where CompositeType = 39
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.

    FROM TABLE
      TCompRet
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: TCompRet scan count 1, logical reads: (regular=11470 apf=0
total=11386), physical reads: (regular=8 apf=392 total=400), apf IOs used=392
Total writes for this command: 0
```

SI Query:

```
1> select * from TCompRet where CompositeType = 39
2> and ClassificationCode = 147
3> union
4> select * from TCompRet where CompositeType = 39
5> and (ClassificationCode < 147
6>  or ClassificationCode > 147)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is INSERT.
        The update mode is direct.

        FROM TABLE
            TCompRet
        Nested iteration.
        Index : ixcompreturn3
        Forward scan.
        Positioning by key.
        Keys are:
            ClassificationCode  ASC
            CompositeType  ASC
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 1
        The type of query is INSERT.
        The update mode is direct.

FROM TABLE
            TCompRet
        Nested iteration.
        Index : ixcompreturn3
        Forward scan.
        Positioning at index start.
        Using I/O Size 2 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.
        Using I/O Size 2 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.
```

```
STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: TCompRet scan count 1, logical reads: (regular=20 apf=0 total=20),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: TCompRet scan count 1, logical reads: (regular=1887 apf=0
total=1887), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=29 apf=0 total=29),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

Example 6.8:

Original Query: select * from TClnt where Charity = 'Y' and Deal = 'Y'

Rule: if Charity = 'Y' then Deal = 'N' (100% confidence)

Original query: 6154 I/O
SI Query: 1 I/O – for the rule page.

This is an ‘inverse’ example, where the 2 predicates in the where clause conflict with a 100% rule, hence no results will be returned.

This example is different in that it uses the SI algorithm to answer the query indirectly by telling that it has no result set. This is because there is a 100% confidence rule that the query's predicate conflicts with.

Original Query:

```
1> select * from TClnt
2> where Charity = 'Y' and Deal = 'Y'
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.

    FROM TABLE
      TClnt
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: TClnt scan count 1, logical reads: (regular=6154 apf=8 total=6162),
physical reads: (regular=8 apf=664 total=672), apf IOs used=664
Total writes for this command: 0
```

Example 6.9:

Original Query: select distinct Security from TQuote where QuoteCode = 'SETTLEMENT'

Rule: if QuoteCode = 'SETTLEMENT' then MarketCode = 'MM' (72%)

Original Query: 58134 I/Os

SI Query: 8724 I/Os

In this example, I/O is reduced in the SI query even though the same access path is used (indexed acccess). In this, SI enables greater selectivity of the index, reducing the I/O required by some 85%.

Original Query:

```
1> select distinct Security from TQuote
2> where QuoteCode = 'SETTLEMENT'
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Executed in parallel by coordinating process and 3 worker processes.
    Worktable1 created for DISTINCT.

    FROM TABLE
      TQuote
    Nested iteration.
    Index : tqu_x1
    Forward scan.
    Positioning by key.
    Keys are:
      QuoteCode  ASC
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 2
```

The type of query is SELECT.
Executed by coordinating process.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: TQuote scan count 3, logical reads: (regular=8424 apf=0 total=8424), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=49710 apf=0 total=49710), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct Security from TQuote
2> where QuoteCode = 'SETTLEMENT'
3> and (MarketCode > 'MM' or MarketCode < 'MM')
4> union
5> select distinct Security from TQuote
6> where QuoteCode = 'SETTLEMENT'
7> and MarketCode = 'MM'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Executed in parallel by coordinating process and 3 worker processes.

```
    FROM TABLE
        TQuote
    Nested iteration.
    Index : tqu_x1
    Forward scan.
    Positioning by key.
    Keys are:
        QuoteCode  ASC
        MarketCode  ASC
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable1.

STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Executed in parallel by coordinating process and 3 worker processes.

    FROM TABLE
        TQuote
    Nested iteration.
    Index : tqu_x1
    Forward scan.
    Positioning by key.
    Keys are:
        QuoteCode  ASC
        MarketCode  ASC
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable1.

STEP 1
```

The type of query is SELECT.
Executed by coordinating process.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

Table: TQuote scan count 3, logical reads: (regular=1681 apf=0 total=1681), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: TQuote scan count 3, logical reads: (regular=6755 apf=0 total=6755), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Chapter 6 Queries – Second Data Set

Example 6.10:

Original Query: select distinct machine_user_name from login_info where login_name = 'PWalds'

Rule: if login_name = 'PWalds' then machine name = 'RD-02727' (85% confidence)

Original Query: 2078 I/Os
SI Query: 1426 I/Os

This is a sample query that would be executed by a Security/Audit group to check the machines that logins are from.

Original Query:

```
1> select distinct machine_user_name
2> from login_info
3> where login_name = 'PWalds'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.

STEP 1
The type of query is INSERT.
The update mode is direct.
Executed in parallel by coordinating process and 3 worker processes.
Worktable1 created for DISTINCT.

FROM TABLE
login_info
Nested iteration.
Index : ix2_login_info
Forward scan.
Positioning by key.
Keys are:
login_name ASC
Executed in parallel with a 3-way hash scan.
Using I/O Size 16 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
Executed by coordinating process.
This step involves sorting.

FROM TABLE
Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: login_info scan count 3, logical reads: (regular=738 apf=0 total=738), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=1340 apf=0 total=1340), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct machine_user_name
2> from login_info
3> where login_name = 'PWalds'
4> and machine_name = 'RD-02727'
5> union
6> select distinct machine_user_name
7> from login_info
8> where login_name = 'PWalds'
9> and (machine_name < 'RD-02727'
10>  OR machine_name > 'RD-02727')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.

STEP 1

The type of query is INSERT.

The update mode is direct.

Executed in parallel by coordinating process and 3 worker processes.

FROM TABLE

login_info

```
    Nested iteration.
    Index : ix2_login_info
    Forward scan.
    Positioning by key.
    Keys are:
      login_name  ASC
      machine_name  ASC
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Executed in parallel by coordinating process and 3 worker processes.

    FROM TABLE
      login_info
    Nested iteration.
    Index : ix2_login_info
    Forward scan.
    Positioning by key.
    Keys are:
      login_name  ASC
    Executed in parallel with a 3-way hash scan.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

STEP 1
    The type of query is SELECT.
    Executed by coordinating process.
    This step involves sorting.
```

```
    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: login_info  scan count 3,  logical reads: (regular=643 apf=0 total=643),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: login_info  scan count 3,  logical reads: (regular=738 apf=0 total=738),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1   scan count 0,  logical reads: (regular=45 apf=0 total=45),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Total writes for this command: 0
```

Example 6.11:

Original Query: select distinct machine_name from login_info where login_name = 'PWalds'

Rule: if login_name = 'PWalds' then machine name = 'RD-02727' (85% confidence)

Original query: 1312 I/Os
SI Query: 197 I/Os

This query is similar to the previous one, but the column in the select list is indexed (whereas in the previous query it is not). Hence this is an ‘index covered’ query, and the improvement can be compared to the non-index covered query above.

Original Query:

```
1> select distinct machine_name
2> from login_info
3> where login_name = 'PWalds'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

login_info

Nested iteration.

Index : ix2_login_info

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

login_name ASC

Using I/O Size 16 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: login_info scan count 1, logical reads: (regular=22 apf=0 total=22),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1   scan count 0, logical reads: (regular=1290 apf=0
total=1290), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct machine_name from login_info
2> where login_name = 'PWalds'
3> and (machine_name < 'RD-02727'
4>   or machine_name > 'RD-02727')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
login_info
Nested iteration.
Index : ix2_login_info
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
login_name ASC
Using I/O Size 16 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.

This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: login_info scan count 1, logical reads: (regular=22 apf=0 total=22),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=175 apf=0 total=175),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

Example 6.12:

Original query: select distinct flow_calc_code from flow
where flow_type_code = 'INTEREST'

Rule: if flow_type_code = 'INTEREST' then flow_calc_code = 'SIMPLE' (99% confidence)

Original Query: 198161
SI Query: 2097

This query is selecting the type of cash flow from a table storing all types of flows. This looks at the calculation type used for Interest based cash flows.

Original Query:

```
1> select distinct flow_calc_code from flow
```

```
2> where flow_type_code = 'INTEREST'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

flow

Nested iteration.

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: flow scan count 1, logical reads: (regular=44849 apf=0 total=44849), physical reads: (regular=0 apf=0 total=0), apf IOs used=0

Table: Worktable1 scan count 0, logical reads: (regular=153312 apf=0 total=153312), physical reads: (regular=0 apf=0 total=0), apf IOs used=0

Total writes for this command: 2

SI Query:

```
1> select distinct flow_calc_code from flow
2> where flow_type_code = 'INTEREST'
3> and (flow_calc_code < 'SIMPLEINT'
4>   or flow_calc_code > 'SIMPLEINT')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

flow

Nested iteration.

Index : ix_flow

Forward scan.

Positioning by key.

Keys are:

flow_calc_code ASC

Using I/O Size 16 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

FROM TABLE

flow

Nested iteration.

Index : ix_flow

Forward scan.

Positioning by key.

Keys are:

flow_calc_code ASC

Using I/O Size 2 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 2 Kbytes for data pages.

```
        With LRU Buffer Replacement Strategy for data pages.

        FROM TABLE
           flow
        Nested iteration.
        Using Dynamic Index.
        Forward scan.
        Positioning by Row IDentifier (RID).
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
           Worktable1.

     STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
           Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable2 is done in Serial

The sort for Worktable1 is done in Serial

Table: flow  scan count 2,  logical reads: (regular=1719 apf=0 total=1719),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: Worktable1   scan count 0,  logical reads: (regular=188 apf=0 total=188),
 physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: Worktable2   scan count 1,  logical reads: (regular=190 apf=0 total=190),
 physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Total writes for this command: 5
```

Example 6.13:

Original Query: select distinct curve_type_code from curve
where currency_code = 'CZK'

Rule: if currency_code = 'CZK' then curve_type_code = 'INTEREST' (100% confidence)

Original Query: 479 I/Os
SI query: 1 I/O (rule covered - assuming a single rule page)

This is a ‘rule covered’ query – the rule used by SI algorithm answers the query completely.

Original Query:

```
1> select distinct curve_type_code from curve
2> where currency_code = 'CZK'
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Worktable1 created for DISTINCT.

    FROM TABLE
      curve
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 2
```

```
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: curve  scan count 1,  logical reads: (regular=394 apf=0 total=394),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: Worktable1   scan count 0,  logical reads: (regular=85 apf=0 total=85),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Total writes for this command: 0
```

Example 6.14:

Original Query: select count(*) from curve where currency_code = 'USD'

Rule: if currency_code = 'USD' then curve_type_code = 'FX' (79% confidence)

Original Query: 394 I/Os
SI Query: 797 I/Os.

This is similar to the previous query but is not rule covered – as the rule is not with 100% confidence.
In this example, SI is actually detrimental to performance.

Original Query:

```
1> select count(*) from curve
2> where currency_code = 'USD'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

Evaluate Ungrouped COUNT AGGREGATE.

FROM TABLE

curve

Nested iteration.

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

STEP 2

The type of query is SELECT.

Table: curve scan count 1, logical reads: (regular=394 apf=0 total=394), physical reads: (regular=0 apf=0 total=0), apf IOs used=0

Total writes for this command: 0

SI Query:

```
1> select count(*) from curve
2> where currency_code = 'USD'
3> and curve_type_code = 'FX'
4> union
5> select count(*) from curve
6> where currency_code = 'USD'
7> and (curve_type_code < 'FX' or curve_type_code >
'FX')
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
      curve
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.

  STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
      Worktable1.

  STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
      curve
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.

  STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
      Worktable1.
```

```
STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: curve  scan count 1,  logical reads: (regular=394 apf=0 total=394),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: curve  scan count 1,  logical reads: (regular=394 apf=0 total=394),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1   scan count 0,  logical reads: (regular=9 apf=0 total=9),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Total writes for this command: 0
```

Example 6.15:

Original Query: select count(*) from trade where trade_status_code = 'MATURED'

Rule: if trade_status_code = 'MATURED' then trade_info_code = 'FX' (90% confidence)

Original Query: 106799 I/Os
SI Query: 2328 I/Os.

This query is based on trades that have matured (expired or settled in the past). The rule shows that 90% of the matured trades are foreign exchange trades, which is because the mature quicker than other types.
SI is shown to provide a huge advantage in reducing the I/O required to answer the query.

Original Query:

```
1> select count(*) from trade
2> where trade_status_code = 'MATURED'
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.

    STEP 1
        The type of query is SELECT.
        Evaluate Ungrouped COUNT AGGREGATE.
        Executed in parallel by coordinating process and 3 worker processes.

        FROM TABLE
            trade
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Executed in parallel with a 3-way hash scan.
        Using I/O Size 16 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.

        Parallel result buffer merge.

    STEP 2
        The type of query is SELECT.
        Executed by coordinating process.

Table: trade  scan count 3,  logical reads: (regular=106799 apf=0 total=106799),
 physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
Total writes for this command: 0
```

SI Query:

```
1> select count(*) from trade
2> where trade_status_code = 'MATURED'
3> and trade_info_code = 'FX'
4> union
5> select count(*) from trade
6> where trade_status_code = 'MATURED'
7> and (trade_info_code < 'FX' or trade_info_code >
   'FX')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

Evaluate Ungrouped COUNT AGGREGATE.

FROM TABLE

trade

Nested iteration.

Index : ix2_trade

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

trade_info_code ASC

trade_status_code ASC

Using I/O Size 16 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

STEP 2

The type of query is INSERT.

The update mode is direct.

TO TABLE

Worktable1.

STEP 1

```
        The type of query is SELECT.
        Evaluate Ungrouped COUNT AGGREGATE.

        FROM TABLE
            trade
        Nested iteration.
        Index : ix2_trade
        Forward scan.
        Positioning at index start.
        Index contains all needed columns. Base table will not be read.
        Using I/O Size 16 Kbytes for index leaf pages.
        With LRU Buffer Replacement Strategy for index leaf pages.

    STEP 2
        The type of query is INSERT.
        The update mode is direct.
        TO TABLE
            Worktable1.

    STEP 1
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 16 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.

Table: trade  scan count 1,  logical reads: (regular=3 apf=0 total=3),  physical
reads: (regular=0 apf=0 total=0),  apf IOs used=0
Table: trade  scan count 1,  logical reads: (regular=2316 apf=0 total=2316),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1   scan count 0,  logical reads: (regular=9 apf=0 total=9),
physical reads: (regular=0 apf=0 total=0),  apf IOs used=0
```

Total writes for this command: 0

Example 6.16:

Original Query: select count(*) from sec where class_name = 'ISwapLeg'

Rule: if class_name = 'ISwapLeg' then sec_def_code = 'SPECIFIC' (93% confidence)

Original query: 8820 I/Os
SI query: 683 I/Os

This query is based on looking at the classifications in a security table.

Original Query:

```
1> select count(*) from sec
2> where class_name = 'ISwapLeg'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.

STEP 1
The type of query is SELECT.
Evaluate Ungrouped COUNT AGGREGATE.
Executed in parallel by coordinating process and 3 worker processes.

FROM TABLE
sec
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Executed in parallel with a 3-way hash scan.
Using I/O Size 16 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

Parallel result buffer merge.

STEP 2

The type of query is SELECT.

Executed by coordinating process.

Table: sec scan count 3, logical reads: (regular=8820 apf=0 total=8820),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select count(*)
2> from sec where class_name = 'ISwapLeg'
3> and sec_def_code = 'SPECIFIC'
4> union
5> select count(*)
6> from sec where class_name = 'ISwapLeg'
7> and (sec_def_code < 'SPECIFIC'
8>   or sec_def_code > 'SPECIFIC')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

Evaluate Ungrouped COUNT AGGREGATE.

FROM TABLE

sec

Nested iteration.

Index : ix1_sec

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

sec_def_code ASC

class_name ASC

```
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
        Worktable1.

STEP 1
    The type of query is SELECT.
    Evaluate Ungrouped COUNT AGGREGATE.

    FROM TABLE
        sec
    Nested iteration.
    Index : ix1_sec
    Forward scan.
    Positioning at index start.
    Index contains all needed columns. Base table will not be read.
    Using I/O Size 16 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

STEP 2
    The type of query is INSERT.
    The update mode is direct.
    TO TABLE
        Worktable1.

STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
```

With MRU Buffer Replacement Strategy for data pages.

Table: sec scan count 1, logical reads: (regular=334 apf=0 total=334),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: sec scan count 1, logical reads: (regular=340 apf=0 total=340),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1 scan count 0, logical reads: (regular=9 apf=0 total=9),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

Example 6.17:

Original Query: select distinct source from sec where class_name = 'ISwapLeg'

Rule: if class_name = 'ISwapLeg' then sec_def_code = 'SPECIFIC' (93% confidence)

Original Query: 121441 I/Os
SI Query: 9808 I/Os

Original Query:

```
1> select distinct source from sec
2> where class_name = 'ISwapLeg'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3 worker processes.

STEP 1
The type of query is INSERT.
The update mode is direct.
Executed in parallel by coordinating process and 3 worker processes.
Worktable1 created for DISTINCT.

FROM TABLE

sec

Nested iteration.

Table Scan.

Forward scan.

Positioning at start of table.

Executed in parallel with a 3-way hash scan.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

Executed by coordinating process.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: sec scan count 3, logical reads: (regular=8858 apf=0 total=8858), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=112583 apf=0 total=112583), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct source from sec
2> where class_name = 'ISwapLeg'
```

```
3> and sec_def_code = 'SPECIFIC'
4> union
5> select distinct source from sec
6> where class_name = 'ISwapLeg'
8> and (sec_def_code < 'SPECIFIC'
9>    or sec_def_code > 'SPECIFIC')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 3 worker processes.

STEP 1

The type of query is INSERT.

The update mode is direct.

Executed in parallel by coordinating process and 3 worker processes.

FROM TABLE

sec

Nested iteration.

Table Scan.

Forward scan.

Positioning at start of table.

Executed in parallel with a 3-way hash scan.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 1

The type of query is INSERT.

The update mode is direct.

Executed by coordinating process.

FROM TABLE

sec

Nested iteration.

Index : ix1_sec

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

```
    Keys are:
        sec_def_code  ASC
        class_name  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

    FROM TABLE
        sec
    Nested iteration.
    Index : ix1_sec
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
        sec_def_code  ASC
        class_name  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.

    FROM TABLE
        sec
    Nested iteration.
    Using Dynamic Index.
    Forward scan.
    Positioning by Row IDentifier (RID).
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable1.

STEP 1
    The type of query is SELECT.
    Executed by coordinating process.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
```

Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

Table: sec scan count 3, logical reads: (regular=8858 apf=0 total=8858),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable2 is done in Serial

Table: sec scan count 3, logical reads: (regular=38 apf=0 total=38), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable2 scan count 1, logical reads: (regular=233 apf=0 total=233),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1 scan count 0, logical reads: (regular=679 apf=0 total=679),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 7

Example 6.18:

Original Query: select distinct data_group_code from auth_status where auth_type_code = 'NEW'

Rule: if auth_type_code = 'NEW' then data_group_code = 'trade_stlmt' (80% confidence)

Original Query: 893681 I/Os
SI query: 114758 I/Os

Original Query:

```
1> select distinct data_group_code
2> from auth_status
3> where auth_type_code = 'NEW'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 3 worker processes.

STEP 1

The type of query is INSERT.

The update mode is direct.

Executed in parallel by coordinating process and 3 worker processes.

Worktable1 created for DISTINCT.

FROM TABLE

auth_status

Nested iteration.

Table Scan.

Forward scan.

Positioning at start of table.

Executed in parallel with a 3-way hash scan.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

Executed by coordinating process.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 16 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Parallel

Table: auth_status scan count 3, logical reads: (regular=56487 apf=0 total=56487), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=837194 apf=0 total=837194), physical reads: (regular=0 apf=0 total=0), apf IOs used=0

Total writes for this command: 3

SI Query:

```
1> select distinct data_group_code
2> from auth_status
3> where auth_type_code = 'NEW'
4> and (data_group_code < 'trade_stlmt'
5>   or data_group_code > 'trade_stlmt')
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

auth_status

Nested iteration.

Index : ix2_auth_status

Forward scan.

Positioning by key.

Keys are:

data_group_code ASC

Using I/O Size 16 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

FROM TABLE

auth_status

Nested iteration.

Index : ix2_auth_status

Forward scan.

Positioning by key.

Keys are:

data_group_code ASC

Using I/O Size 2 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

FROM TABLE
auth_status
Nested iteration.
Using Dynamic Index.
Forward scan.
Positioning by Row IDentifier (RID).
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 16 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable2 is done in Serial

The sort for Worktable1 is done in Serial

Table: auth_status scan count 3, logical reads: (regular=16405 apf=0 total=16405), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=1043 apf=0 total=1043), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable2 scan count 1, logical reads: (regular=97310 apf=0 total=97310), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 541

Example 6.19:

Original Query: select spot_date from trade where process_org_id = 3

Rule: if process_org_id = 3 then subject_org_id = 1 (100% confidence)

Original query: 78373 I/Os
SI query: 49316 I/Os

In this example, SI enables a change to the access path, giving some improvement in the efficiency of execution.

Original Query:

```
1> select spot_date from trade
2> where process_org_id = 3
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SET STATUS ON.

Total writes for this command: 0

QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is SELECT.

    FROM TABLE
      trade
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 16 Kbytes for data pages.
```

With LRU Buffer Replacement Strategy for data pages.

Table: trade scan count 1, logical reads: (regular=78373 apf=0 total=78373),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select spot_date from trade
2> where process_org_id = 3
3> and subject_org_id = 1
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SET STATUS ON.

Total writes for this command: 0

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 3 worker processes.

STEP 1
The type of query is SELECT.
Executed in parallel by coordinating process and 3 worker processes.

FROM TABLE
trade
Nested iteration.
Index : ix3_trade
Forward scan.
Positioning by key.
Keys are:
subject_org_id ASC
process_org_id ASC
Executed in parallel with a 3-way hash scan.
Using I/O Size 16 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 16 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

Parallel network buffer merge.

Table: trade scan count 3, logical reads: (regular=49316 apf=0 total=49316),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

Chapter 7 Queries

Example 7.1:

With data distribution such that the antecedent is at the lowest end of the normal distribution curve:

Data distribution:

```
select count(*), subject_type from titles
group by subject_type
order by 1
```

Total	Subject
60	Astronomy
60	Media
120	Astrology
120	Health
250	Design
250	Travelling
500	Geography
500	Sociology
1000	Chemicals

1000	Gardening
2500	Business
2500	Economics
5000	Beauty
5000	History
10000	Biology
10000	Plants
50000	Languages
50000	Science
100000	Maths
100000	Music
150000	Art

Original Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Using Clustered Index.

Index : Cl_1x

Forward scan.

Positioning by key.

Keys are:

subject_type ASC

Using I/O Size 2 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

```
    TO TABLE
        Worktable1.

  STEP 2
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=11 apf=0 total=11), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=68 apf=0 total=68),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

SI Query:

From the rule, we know that:

```
if subject_type = 'Astronomy' then price = 29.95 (70% confidence)
```

Hence the corresponding SI query only asks for the information requested by the original query and unknown from the rule.

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
3> and (price < 29.95 or price > 29.95)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is INSERT.
        The update mode is direct.
        Worktable1 created for DISTINCT.

        FROM TABLE
            titles
        Nested iteration.
        Using Clustered Index.
        Index : Cl_1x
        Forward scan.
        Positioning by key.
        Keys are:
            subject_type  ASC
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=11 apf=0 total=11), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=26 apf=0 total=26),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
```

Total writes for this command: 0

From this example, we can see that the SI query is more efficient than the original query by 42 fewer I/Os (79 I/Os for the original and 37 I/Os for the SI query), which is 46% of the original query's I/O – over a 50% improvement.

Example 7.2:

With data distribution changed so that antecedent is at the high or top end of the normal distribution curve:

When the data is changed so that `subject_type = 'Astronomy'` is at the top end of the normal distribution, as follows:

```
select count(*), subject_type from titles
group by subject_type
order by 1
```

Total	Subject
60	Art
60	Media
120	Astrology
120	Health
250	Design
250	Travelling
500	Geography
500	Sociology
1000	Chemicals
1000	Gardening
2500	Business
2500	Economics
5000	Beauty

5000	History
10000	Biology
10000	Plants
50000	Languages
50000	Science
100000	Maths
100000	Music
150000	Astronomy

Original Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Using Clustered Index.

Index : Cl_1x

Forward scan.

Positioning by key.

Keys are:

subject_type ASC

Using I/O Size 2 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=6533 apf=0 total=6533),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=152696 apf=0
total=152696), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 2

SI Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
3> and (price < 29.95 or price > 29.95)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Using Clustered Index.
Index : Cl_1x
Forward scan.
Positioning by key.

```
    Keys are:
      subject_type  ASC
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 2
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
      Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=6533 apf=0 total=6533),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=26613 apf=0
total=26613), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

In this case, where the antecedent is at the high end of the normal distribution, I/O is reduced by 126079 I/Os. This is 20% of the I/O of the original query.
This is because the antecedent, being at the top end of the normal distribution curve, has had its selectivity increased significantly enough to have made a difference, whereas when the antecedent was at the low end of the normal distribution, selectivity was high to start with, hence adding the SI did not increase selectivity by the same magnitude.

Example 7.3:

Here the data is changed so that the antecedent is at neither the top end nor the bottom end of the normal distribution – but at the lower-mid end, as can be seen, `where subject_type = 'Astronomy'`.

With data distribution changed so that antecedent is at the lower-mid end of the normal distribution curve:

```
select count(*), subject_type from titles
group by subject_type
order by 1
```

Total	Subject
60	Art
60	Media
120	Astrology
120	Health
250	Design
250	Travelling
500	Geography
500	Sociology
1000	Chemicals
1000	Gardening
2500	Astronomy
2500	Business
5000	Beauty
5000	History
10000	Biology
10000	Plants
50000	Languages
50000	Science
100000	Maths
100000	Music
150000	Economics

Original Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Using Clustered Index.
Index : Cl_1x
Forward scan.
Positioning by key.
Keys are:
subject_type ASC
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=116 apf=0 total=116),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=2550 apf=0 total=2550),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
3> and (price < 29.95 or price > 29.95)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Using Clustered Index.
Index : Cl_1x
Forward scan.
Positioning by key.
Keys are:
subject_type ASC
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=116 apf=0 total=116),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=770 apf=0 total=770),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

Here, the benefit is also profound – I/O for the SI query is reduced to just over 33% of that of the original query.

Example 7.4:

With the data distribution changed again so that the antecedent is at the middle-upper range on the normal distribution curve:

```
select count(*), subject_type from titles
group by subject_type
order by 1
```

Total	Subject
60	Art
60	Media
120	Astrology
120	Health
250	Design

250	Travelling
500	Geography
500	Sociology
1000	Chemicals
1000	Gardening
2500	Business
2500	Plants
5000	Beauty
5000	History
10000	Astronomy
10000	Biology
50000	Languages
50000	Science
100000	Maths
100000	Music
150000	Economics

Original Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Using Clustered Index.

Index : testx

Forward scan.

Positioning by key.
Keys are:
subject_type ASC
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=431 apf=0 total=431),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=10185 apf=0
total=10185), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct price from titles
2> where subject_type = 'Astronomy'
3> and (price < 29.95 or price > 29.95)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.

```
        The update mode is direct.
        Worktable1 created for DISTINCT.

        FROM TABLE
            titles
        Nested iteration.
        Using Clustered Index.
        Index : testx
        Forward scan.
        Positioning by key.
        Keys are:
            subject_type  ASC
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=431 apf=0 total=431),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=3059 apf=0 total=3059),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

Here I/O is reduced to less than 33% of the original query.

From the previous 4 examples, we can see that the higher on the normal distribution curve that the antecedent is, the greater the benefit of SI in reducing I/O by a higher proportion.

The following examples are based on the rule:

```
if title = 'Maths for beginners' then price = 15.00
(70% confidence)
```

The rule's antecedent is first at the low end of the normal distribution, then the data is changed so that it is at the high end with examples included in the intermediate positions on the normal distribution curve.

Example 7.5:

With data distribution changed so that antecedent is at the low end of the normal distribution curve:

```
select count(*), title from titles
group by title
order by 1
```

Total	Title
60	Maths for beginners
60	World Discovery
125	European Cities
125	Houses and Gardens
250	Cats and Dogs
250	Zoo Animals
500	House Plants
500	Make Up Colour

1000	Australia
1000	PC World
2000	Running
2000	Yoga for All
3500	Internet Design
3500	Starting on the Internet
6000	Holistic Health
6000	Operating Systems
10000	Java Beans
10000	Networks
20000	Horticulture
20000	Jewellery Design
25000	Gardening
25000	Refloxology for Hands
50000	Algebra
50000	Style
75000	Advanced Maths
75000	Basic Grammar
100000	Costumes

Original Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Index : I_x2
Forward scan.
Positioning by key.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=9 apf=0 total=9), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=68 apf=0 total=68), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
```

```
5> select distinct total_sold from titles
6> where title = 'Maths for beginners'
7> and price = 15
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is INSERT.
    The update mode is direct.

    FROM TABLE
      titles
    Nested iteration.
    Index : I_x2
    Forward scan.
    Positioning by key.
    Keys are:
      title  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 1
    The type of query is INSERT.
    The update mode is direct.

    FROM TABLE
      titles
    Nested iteration.
    Index : I_x2
    Forward scan.
    Positioning by key.
    Keys are:
      title  ASC
      price  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
```

```
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
       Worktable1.

  STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
       Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: titles scan count 1, logical reads: (regular=9 apf=0 total=9), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: titles scan count 1, logical reads: (regular=7 apf=0 total=7), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=68 apf=0 total=68),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

In this example, there is no advantage to using SI. Selectivity was high in the original query, and the SI query did not add sufficient extra selectivity that could reduce I/O. Also the columns selected were not included in the index hence access to underlying data pages was necessary.

Example 7.6:

This is another example query based on the same data and rule.

Original Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Index : testx2

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

title ASC

Using I/O Size 2 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 2 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=7 apf=0 total=7), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=68 apf=0 total=68), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : testx2
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

```
    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=7 apf=0 total=7), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=26 apf=0 total=26),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

With the antecedent at the low end, the SI query has reduced I/O to 44% of the original query – over 50% improvement.

With data distribution changed so that antecedent is at the highest end of the normal distribution curve:

```
select count(*), title from titles
group by title
order by 1 desc
```

Total	Title
100000	Maths for beginners
75000	Advanced Maths
75000	Basic Grammar
50000	Algebra
50000	Style
25000	Gardening
25000	Refloxology for Hands
20000	Horticulture

20000	Jewellery Design
10000	Java Beans
10000	Networks
6000	Operating Systems
6000	Holistic Health
3500	Internet Design
3500	Starting on the Internet
2000	Running
2000	Yoga for All
1000	Australia
1000	PC World
500	House Plants
500	Make Up Colour
250	Cats and Dogs
250	Zoo Animals
125	European Cities
125	Houses and Gardens
60	Costumes
60	World Discovery

Example 7.7:

Original Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : I_x2
Forward scan.
Positioning by key.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=6638 apf=0 total=6638),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=101199 apf=0
total=101199), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct total_sold from titles
```

```
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
5> select distinct total_sold from titles
6> where title = 'Maths for beginners'
7> and price = 15
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

FROM TABLE

titles

Nested iteration.

Index : I_x2

Forward scan.

Positioning by key.

Keys are:

title ASC

Using I/O Size 2 Kbytes for index leaf pages.

With LRU Buffer Replacement Strategy for index leaf pages.

Using I/O Size 2 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 1

The type of query is INSERT.

The update mode is direct.

FROM TABLE

titles

Nested iteration.

Index : I_x2

Forward scan.

Positioning by key.

Keys are:

```
        title  ASC
        price  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
        Worktable1.

STEP 1
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

Table: titles scan count 1, logical reads: (regular=6638 apf=0 total=6638),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: titles scan count 1, logical reads: (regular=4639 apf=0 total=4639),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=101199 apf=0
total=101199), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

This gives no improvement in I/O.

Example 7.8

Original Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : I_x2
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=2277 apf=0 total=2277),

physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=101802 apf=0
total=101802), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 1

SI Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : I_x2
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED

Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=2277 apf=0 total=2277),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=30500 apf=0
total=30500), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

This gives a large improvement in I/O – 69664 fewer I/Os – which is only 31% of the I/Os of the original query, due to increased selectivity enabling increase in the use of the index.

With the data distribution changed so that the antecedent is on the lower-mid range of the normal distribution

```
select count(*), title from titles
group by title
order by 1
```

Total	Title
60	Running
60	World Discovery
125	European Cities
125	Houses and Gardens
250	Cats and Dogs
250	Zoo Animals
500	House Plants
500	Make Up Colour

1000	Australia
1000	PC World
2000	Maths for beginners
2000	Yoga for All
3500	Internet Design
3500	Starting on the Internet
6000	Holistic Health
6000	Operating Systems
10000	Java Beans
10000	Networks
20000	Horticulture
20000	Jewellery Design
25000	Gardening
25000	Refloxology for Hands
50000	Algebra
50000	Style
75000	Advanced Maths
75000	Basic Grammar
100000	Costumes

Example 7.9:

Original Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : I_x2
Forward scan.
Positioning by key.
Keys are:
title ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=722 apf=0 total=722),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=2029 apf=0 total=2029),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct total_sold from titles
```

```
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
5> select distinct total_sold from titles
6> where title = 'Maths for beginners'
7> and price = 15
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Worktable1 created for DISTINCT.

    FROM TABLE
      titles
    Nested iteration.
    Index : I_x2
    Forward scan.
    Positioning by key.
    Keys are:
      title  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 2
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
      Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
```

Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=722 apf=0 total=722),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: titles scan count 1, logical reads: (regular=110 apf=0 total=110),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1 scan count 0, logical reads: (regular=2029 apf=0 total=2029),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

Example 7.10:

Original Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Index : I_x2
Forward scan.
Positioning by key.
Index contains all needed columns. Base table will not be read.
Keys are:
title ASC

Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=89 apf=0 total=89), physical reads: (regular=8 apf=129 total=137), apf IOs used=81
Table: Worktable1 scan count 0, logical reads: (regular=2041 apf=0 total=2041), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

```
    FROM TABLE
        titles
    Nested iteration.
    Index : I_x2
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
        title  ASC
    Using I/O Size 2 Kbytes for index leaf pages.
    With LRU Buffer Replacement Strategy for index leaf pages.
    TO TABLE
        Worktable1.

  STEP 2
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE
        Worktable1.
    Using GETSORTED
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=89 apf=0 total=89), physical
reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=820 apf=0 total=820),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

SI has reduced I/O to 42% of that required by the original query.

When the antecedent is changed so that it is at the mid-higher end of the normal distribution curve:

```
select count(*), title from titles
group by title
order by 1
```

Total	Title
60	Costumes
60	World Discovery
125	European Cities
125	Houses and Gardens
250	Cats and Dogs
250	Zoo Animals
500	House Plants
500	Make Up Colour
1000	Australia
1000	PC World
2000	Running
2000	Yoga for All
3500	Internet Design
3500	Starting on the Internet
6000	Holistic Health
6000	Operating Systems
10000	Java Beans
10000	Networks
20000	Jewellery Design
20000	Maths for beginners
25000	Gardening
25000	Refloxology for Hands
50000	Algebra
50000	Style
75000	Advanced Maths
75000	Basic Grammar
100000	Horticulture

Example 7.11:

Original Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.
Worktable1 created for DISTINCT.

FROM TABLE
titles
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 2
The type of query is SELECT.
This step involves sorting.

FROM TABLE
Worktable1.
Using GETSORTED
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=21289 apf=0 total=21289),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=20245 apf=0
total=20245), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct total_sold from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
4> union
5> select distinct total_sold from titles
6> where title = 'Maths for beginners'
7> and price = 15
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is INSERT.
The update mode is direct.

FROM TABLE
titles
Nested iteration.
Table Scan.
Forward scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
TO TABLE
Worktable1.

STEP 1
The type of query is INSERT.
The update mode is direct.

```
        FROM TABLE
            titles
        Nested iteration.
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With LRU Buffer Replacement Strategy for data pages.
        TO TABLE
            Worktable1.

    STEP 1
        The type of query is SELECT.
        This step involves sorting.

        FROM TABLE
            Worktable1.
        Using GETSORTED
        Table Scan.
        Forward scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes for data pages.
        With MRU Buffer Replacement Strategy for data pages.


Table: titles scan count 1, logical reads: (regular=21289 apf=0 total=21289),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: titles scan count 1, logical reads: (regular=21289 apf=0 total=21289),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
The sort for Worktable1 is done in Serial

Table: Worktable1  scan count 0, logical reads: (regular=20245 apf=0
total=20245), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

Example 7.12:

Original Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is INSERT.

The update mode is direct.

Worktable1 created for DISTINCT.

FROM TABLE

titles

Nested iteration.

Index: I_x2

Forward scan.

Positioning by key.

Index contains all needed columns. Base table will not be read.

Keys are:

Title ASC

Using I/O Size 2 Kbytes for data pages.

With LRU Buffer Replacement Strategy for data pages.

TO TABLE

Worktable1.

STEP 2

The type of query is SELECT.

This step involves sorting.

FROM TABLE

Worktable1.

Using GETSORTED

Table Scan.

Forward scan.

Positioning at start of table.

Using I/O Size 2 Kbytes for data pages.

With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=712 apf=0 total=712),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1 scan count 0, logical reads: (regular=20365 apf=0
total=20365), physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0

SI Query:

```
1> select distinct price from titles
2> where title = 'Maths for beginners'
3> and (price < 15 or price > 15)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
  STEP 1
    The type of query is INSERT.
    The update mode is direct.
    Worktable1 created for DISTINCT.

    FROM TABLE
      titles
    Nested iteration.
    Index: I_x2
    Forward scan.
    Positioning by key.
    Index contains all needed columns. Base table will not be read.
    Keys are:
        Title ASC
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    TO TABLE
      Worktable1.

  STEP 2
    The type of query is SELECT.
    This step involves sorting.

    FROM TABLE

```
    Worktable1.
  Using GETSORTED
  Table Scan.
  Forward scan.
  Positioning at start of table.
  Using I/O Size 2 Kbytes for data pages.
  With MRU Buffer Replacement Strategy for data pages.

The sort for Worktable1 is done in Serial

Table: titles scan count 1, logical reads: (regular=712 apf=0 total=712),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Table: Worktable1  scan count 0, logical reads: (regular=6113 apf=0 total=6113),
physical reads: (regular=0 apf=0 total=0), apf IOs used=0
Total writes for this command: 0
```

In this example, I/O required by the SI query has been reduced to 33% of the I/Os of the original query.

Appendix A5

This appendix provides independent costing of SI using the established Decomposition Algorithm [50] for query processing. It looks at original queries along with their corresponding SI queries in conjunction with the Decomposition Algorithm for costing and comparison.

Processing the Query

The following query is used for demonstrating the Decomposition Algorithm.

Example:

```
SELECT A.b, C.i
FROM table1 A, table2 B, table3 C
WHERE A.a = B.a
AND  B.e = C.e
AND  A.a = value_1
```

Processing the query in the intuitive way of forming a cartesian product, determining which tuples satisfy the $f(A_{(selection\ a = 'value')} * B * C) = true$ condition, and performing a projection on the restricted subset is highly expensive. This is because the cardinality of the product is equal to the product of the cardinalities of $A_{(selection\ a = 'value')}$, B and C. Therefore, query processing algorithms attempt to make this more efficient.

Overview of Decomposition

The Decomposition Algorithm for query processing is called so because it is based on reducing a multi variable query (in this case the variables are *A*, *B* and *C* with corresponding ranges *table1*, *table2* and *table3*, respectively) into smaller

single variable queries. Therefore it *decomposes* or breaks down the query until it cannot be reduced any further.

This is done by two types of operation, which together, can decompose any query completely:

1. *Tuple substitution* - this is where a query with *n* variables is replaced by a query with (*n-1*) variables. This is done by replacing one of the variables with each individual tuple comprising the range of the variable.

2. *Detachment* - this is the process where a query with an overlapping variable is replaced by two sub-queries, such that each has a single variable in common. *Restriction*, or the application of predicates, and *projection* are special cases of detachment.

Tuple substitution should only be used when detachment operations cannot be used to reduce the query. The reason is that the range of variables should be as small as possible by applying detachment before applying tuple substitution, because by applying tuple substitution, the cost of processing the rest of the query is multiplied by the cardinality of the variable that is to be substituted for (tuple by tuple). Therefore this cardinality should be minimised prior to tuple substitution. Hence, as much detachment as possible is performed before using tuple substitution.

The Decomposition Algorithm is made up of four steps, or sub algorithms. The first is *reduction*. This breaks a query into irreducible components. This is then passed to the second step, *sub-query sequencing*, which generates a succession of sub-queries, by using the result of reduction, and passes them, one by one, to the third step, *tuple substitution*. Tuple substitution manages the process of substituting a variable by each tuple in its range. It calls *variable selection*, the final step, in order to choose the variable to substitute for. Variable selection chooses the variable with minimum estimated cost to use for tuple substitution. For this, it may pre-process restriction operations which are one-variable clauses.

The Decomposition Query Processing Algorithm and the Original Query

This section gives a practical in-depth application of the Ingres Decomposition Algorithm for query processing, using SQL instead of QUEL, by applying it to the original query. The next section applies it to the corresponding SI query to allow for subsequent comparison. Hence this provides an independent costing mechanism for comparing the effect of SI on query processing.

The original query, without using association rules and SI, is:

```
SELECT A.b, C.i
FROM table1 A, table2 B, table3 C
WHERE A.a = B.a
AND  B.e = C.e
AND  A.a = value_1
```

The first step of decomposition is reduction - or breaking the query into irreducible components.

An irreducible component of a query is defined by the query having no *disjoint* sub-query and being *one-free*. A disjoint sub-query is a part of the query that can be broken off from the rest of the query, having no variables in common with the rest of the query. One-free means the query has no sub-queries or components with only one variable overlapping (or in common) with a variable in the rest of the query. This query is one-free.

The query must be broken into irreducible components before going any further.

The algorithm for reduction is (paraphrased from [50]):

IF (number of variables in query) > 1 THEN
    IF (query is connected) THEN
        separate into irreducible components
    ELSE

separate into disjoint components

ELSE

no reduction required

From this algorithm, we need a method to tell if a query is connected. This is called a *connectivity* algorithm. The Decomposition query processing strategy provides us with this procedure by using the concept of an incidence matrix. This is a method of representing a query using a matrix with a row for each predicate in the *where* clause, plus a row for the projection list, and a column for each variable in the query.

The incidence matrix for the original query is defined in Figure A5.1:

```
                          A    B    C
T  : A.b,  C.i            1    0    1
C1 : A.a = B.a            1    1    0
C2 : B.e = C.e            0    1    1
C3 : A.a = value_1        1    0    0
```

Figure A5.1 – Incidence Matrix

The *1* digit in the incidence matrix implies the presence of the variable in the clause on the left-hand side, a *0* implies the absence of the variable in the clause.

For each variable (column 1 to column *n*, where *n* is 3 in this example), the logical OR of all rows with a 1 for the variable (or column) is formed. This replaces the first row with the occurrence of a 1 in the column. The rest of the rows with a 1 in the column are deleted.

This is the procedure for the connectivity algorithm. Applying this to the above incidence matrix, we successively get:

First do for the presence of *1* in column one:

	A	B	C
T, C1, C3 :	1	1	1
C2 :	0	1	1

Next, do for the presence of *1* in column two:

T, C1, C3, C2 :	1	1	1

If the final matrix has only one row at the end of applying the algorithm, then it is connected, as above. If there is more than one row, then the query is disjoint, and the connected components of the disjoint query are in the same row of the final matrix. Each row of the final matrix represents a disjoint sub-query.

Now that we know that the original query is a connected, multi-variable query, it needs to be reduced, if possible, into irreducible components. The connectivity algorithm again can be used for this. A query is irreducible if the elimination of any variable causes the query to be disconnected. Such a variable, whose elimination disconnects a query, is called a joining variable. Hence, if a query has no joining variable it is irreducible - fulfilling the reduction stage of the Decomposition Algorithm.

To break this query into irreducible components, we need to check for each variable being a joining variable. We use the connectivity algorithm for this because a variable is joining if its *removal* causes the query to be disconnected. After applying this - removing each variable (column) successively, and testing for connectedness - the query *Q* can be represented by the irreducible components, in a similar matrix, using variables for columns and the irreducible components as rows. From this we generate a *reduced incidence matrix.*

An irreducible component of *Q* is comprised of 1 or more rows of the original query incidence matrix, and is represented by the logical OR of those rows.

Using the incidence matrix defined in Figure A5.1, the query is irreducible because there is no variable whose elimination would disconnect the query.

The reduced incidence matrix is:

	A	B	C
C3 :	1	0	0
C1 :	1	1	0
C2, T:	1	1	1

This is done by organising the rows so that the single-variable clauses, if any, are first, excluding the target list. Then rows that do not contain the target list are listed. The target list is always last.

Once the reduction stage is complete, the output is sent to the second stage, sub-query sequencing. This forms sub-queries from the rows in the reduced-incidence matrix to pass to the tuple substitution stage.

Sub-query sequencing is relatively simple. It takes the first multi-variable row of the reduced incidence matrix, and combines it with one-variable clauses in the same variables. This means combining C1 with C3.

Hence, the sub-queries are:

```
Q1 : C1, C3
Q2 : C2, T
```

Or in SQL query form:

Q1 is:

```
SELECT A.b, B.e INTO table_temp
FROM table1 A, table2 B
WHERE (A.a = B.a)
```

```
AND (A.a = value_1)
```

Q2 is:

```
SELECT (X.b, C.i)
FROM table_temp X, table3 C
WHERE  (X.e = C.e)
```

Q2 uses the output of *Q1*.

For each sub-query, in this case two, *Q1* and *Q2*, the tuple substitution step processes each one. For this it calls variable selection, in order to determine the best variable in the now irreducible query that should be substituted for, tuple by tuple, or row by row.

The first query, *Q1*, is a two-variable query. When passed to tuple substitution, and a variable is selected by variable selection, it becomes a single variable query for each tuple in the second variable that is having its range substituted by a value. Each such query (number of such queries is equal to the number of tuples in the range of the substituted variable) is passed to reduction, returning the result. The larger the range of the variable to be substituted for, the greater the number of single-variable tuple substituted 'reduced' queries that will need to be executed. Results from all 'tuple reduced' queries are concatenated to form the final result.

Variable selection aims to optimise the query by choosing the optimal, or lowest cost variable to substitute for. The variable that is chosen for tuple substitution should have its range reduced as much as possible, by applying query predicates on the variable where possible. The fewer the number of tuples to substitute for tuple by tuple, the smaller the number of reduced tuple substituted queries to execute via the decomposition process again. Hence, a variable with a small range should be chosen and reduced where possible. This can be done for the original query, because there is one single-variable predicate that reduces the range of the variable *A*. However, more variables can have their range reduced in

the SI query, rather than just that of variable *A*.

Reducing the variable reduces the query processing cost. If a variable, say X_i, in a query Q, is chosen for tuple substitution, then the new tuple substituted query is denoted by $Q_i(t)$, where t represents a tuple. There is a query for each substituted tuple from X_i. Let $C(Q)$ denote the cost of processing a query. Then the cost of processing a tuple substituted query is $C(Q_i(t))$. The variable from the query that is selected for tuple substitution should correspond to the i which minimises an estimated value for:

$$C = \sum_{t \in Ri} C(Q_i(t))$$

where t ranges over the tuples of R_i.

The variable that is selected for substitution should minimise this cost. Hence, the optimisation that is performed by variable selection is in minimising cost. If the cost of processing a tuple substituted query is $C(Q_i(t))$, and if this is taken to be independent of the tuple, t, and of the variable i, then the minimum cost, C_i, corresponds to the smallest range R of the variable substituted. Hence we try to reduce the range R of a variable by applying predicates. This is the variable selection method in the query processing that is used in Ingres.

Using this, taking Q_1 into account,

Q_1 is:

```
SELECT A.b, B.e
INTO table_temp
FROM table1 A, table2 B
WHERE (A.a = B.a)
AND (A.a = value_1)
```

Let *table1* have *n1* records and *table2* have *n2* records. If *n1* < *n2*, then *table1*

has the smallest range. In addition, the variable, *table1* can be reduced because there is a single-variable predicate to restrict it. Hence, *table1* should be chosen for tuple substitution by the variable selection process according to the cost minimisation algorithm of Ingres. If *table1* has *d* distinct values, this will generate approximately *n1/d* sub-queries $(Q_1(t))$ - one for each tuple in *table1* assuming the values are evenly distributed.

Applying the predicate on *table1* gives us *n1/d* records left for the tuple substituted queries, as opposed to *n2* queries if *table2* was used, and hence would be generated by substituting for *table2*. Each such sub-query is now single variable. It has only the variable table *table2*. The table, *table1* has been replaced by its actual record values, each one generating a query. The query cost would be:

$$\sum_{i=1}^{n1/d} C(Q_i(t))$$

The smaller the value of *n1*, and the more selective the predicate on it (the higher the value of *d*) then the lower the total query cost.

However, if *n2* < *n1*, and the single variable predicate is on *table1,* then the variable used for tuple substitution should be whichever is less out of *n1/d* or *n2*. In this case, if *n2* is still less than *n1/d,* then *table2* would be used for tuple substitution rather than *table1*. Nonetheless, in the next section it is shown that the cost would still be less for the SI query than for the original query, because of the additional predicate increasing the filtering in the resulting single variable query, with *table1*.

If *table1* has 30 records and *table2* has 5000 records, then *table1* has the smallest range and additionally the variable, *table1,* can be reduced because there is a single-variable predicate to reduce it. Hence, *table1* should be chosen for tuple substitution by the variable selection process according to the cost minimisation algorithm of Ingres. If *table1* has 3 distinct values, this will generate

approximately 10 sub-queries *($Q_1(t)$)* - one for each tuple in *table1* assuming the values are evenly distributed, so that applying the predicate on *table1* gives us 10 records left from it in the query (as opposed to 5000 tuple substituted queries that would be generated by substituting for *table2*). Each such sub-query is now single variable. It has only the table *table2*. *Table1* has been replaced by actual tuple values in place of the table. For example, if the first record in *table1* has the values:

a = '3' and *b = '5'*

then the first tuple substituted query becomes :

```
SELECT B.e, '5'
INTO table_temp
FROM table2 B
WHERE (B.a = '3')
```

The query, *($Q_1(t)$),* will be executed for each of the 10 tuples in *table1*, that have *a = '3'* and the results concatenated to produce a list of *B.e* and *A.b* that are held in *table_temp* table, to be used in Q_2.

After Q_1 is processed by applying tuple substitution, the next sub-query generated by sub-query sequencing, Q_2, is passed into tuple substitution for processing.

Q_2 is:

```
SELECT (X.b, C.i )
FROM table_temp X, table3 C
WHERE  (X.e = C.e)
```

If there are 5000 records in *table_temp* table, which is passed into Q_2, and the *table3* table has 500 records in it, then the *table3* table will be chosen for tuple substitution in order to get each student name and course name. As there are no

predicates to reduce the range of either variable, this will require 500 tuple-substituted queries - one for each *table3* table entry.

So, the complete query requires 10 tuple-substituted sub-queries for Q_1, and 500 for Q_2, totalling 530 single variable queries. Using the Ingres minimum cost estimate for processing the query, this is the cheapest method for the original unmodified query.

The Decomposition Query Processing Algorithm and the Semantically Inequivalent Query

The SI query that results from applying the SI algorithm to the original query is:

```
SELECT (A.b, C.i)
FROM   table1 A, table2 B, table3 C
WHERE  A.a = B.a
AND    B.e = C.e
AND    A.a = value_1
AND    B.f = value_2
```

This uses the rule that given the value of *a*, the value of *f* can be determined.

Passing this through the Decomposition Algorithm for query processing, this query is first broken into irreducible components by reduction. To perform this, the incidence matrix needs to be generated for the SI query. This is:

	A	B	C
T : A.b, C.i	1	0	1
C1 : A.a = B.a	1	1	0
C2 : B.e = C.e	0	1	1
C3 : A.a = value_1	1	0	0
C4 : B.f = value_2	0	1	0

This has two additional rows compared to the original query - for the additional predicates that apply the rules to the tables.

Applying the connectivity algorithm, where each column is taken, and the logical OR of all rows with a 1 in the column is formed, replacing the first row with a 1 in the column, and deleting the rest, we successively get:
(for 1 in column one) :

	A	B	C
T, C1, C3 :	1	1	1
C2 :	0	1	1
C4 :	0	1	0

(for 1 in column 2) :

	A	B	C
T, C1, C2, C3, C4 :	1	1	1

(for 1 in column 3) :

	A	B	C
T, C1, C2, C3, C4 :	1	1	1

The final matrix has one row, hence the query is connected, and has no disjoint components.

Since this is a connected, multi-variable query it needs to be reduced into irreducible components. The query can be reduced if the elimination of any variable causes the incidence matrix to become disconnected. This query has no such joining variable (the removal of no single variable is enough to disconnect the incidence matrix), hence it is irreducible, fulfilling the reduction stage of the Decomposition Algorithm.

To generate the reduced-incidence matrix to pass to sub-query sequencing, the original matrix is re-organised, placing the single variable rows first, and the target list last. The reduced- incidence matrix is therefore:

	A	B	C
C3 :	0	0	1
C4 :	0	1	0
C1 :	0	1	1
C2, T	1	1	1

This reduced-incidence matrix is passed into sub-query sequencing, where sub-queries are formed based on this matrix, to pass to tuple substitution.

When passed to sub-query sequencing, the first multi-variable row from the reduced-incidence matrix is taken and combined with one-variable clauses in the same variables as in the multi-variable clause.

Applying this, the sub-queries generated are:

```
Q1 : C1, C3, C4
Q2 : C2, T
```

In SQL query form, these are:

Q1 is:

```
SELECT INTO table_temp (A.b, B.e)
FROM table1 A, table2 B
WHERE (A.a = B.a)
AND (A.a = value_1)
AND (B.f = value_2)
```

Q2 is:

```
SELECT (X.b, C.i)
FROM table_temp X, table3 C
```

```
WHERE  (X.e = C.e)
```

where *Q2* uses the output of *Q1* via the intermediary table, *table_temp*.

Each sub-query is passed to tuple substitution, where it is processed, and variable selection is invoked to determine which variable to tuple substitute for.

When variable selection is called from tuple substitution, it will optimise the query by choosing the lowest cost variable to replace by tuple substitution. For this, it will reduce the range of variables as much as possible by applying the single-variable query predicates on the variable, to help determine how efficient tuple substitution for the variable would be. It tries to choose a variable with as small a range as possible, because the fewer the tuples in the range, the fewer the number of tuple substituted single-variable queries there are to execute via the decomposition process. With the additional predicates, derived from applying the rules added to the original query to produce the SI query, the range of the variables can be reduced, resulting in fewer tuple-substituted queries, $Q_i(t)$, to execute. The cost of each one of these queries is $C(Q_i(t))$, and the variable that we choose should minimise the cost for substituting by reducing and selecting the variable which minimises it :

$$C = \sum_{i=1}^{R} (Q_i(t))$$

which corresponds to the smallest range of the substituted variable.

Taking *Q1* into account:

Q1 is:

```
SELECT A.b, B.e
INTO table_temp
FROM table1 A, table2 B
WHERE (A.a = B.a)
```

```
AND   (A.a = value_1)
AND   (B.f = value_2)
```

Again, let *table1* have *n1* records and *table2* have *n2* records. If *n1* < *n2*, then *table1* has the smaller range. The variable, *table1* can be reduced because there is a single-variable predicate to restrict it. However, *table2* can also be reduced now with the SI query, given the predicate that SI has added to it. If *table1* has *d1* distinct values for attribute *a*, this will generate approximately *n1/d1* sub-queries $(Q_1(t))$ - one for each tuple in *table1* assuming the values are evenly distributed, if used for tuple substitution.

If *table2* has *d2* distinct values for attribute *f*, this would generate approximately *n2/d2* sub-queries $(Q_1(t))$ - one for each tuple in *table2* again assuming the values are evenly distributed.

If the table, *table1*, has been replaced by its actual record values, each one generating a query. The query cost would be:

$$\sum_{i=1}^{n1/d1} C(Q_i(t))$$

The smaller the value of *n1*, and the more selective the predicate on it (the higher the value of *d1*), then the lower the total query cost would be.

If *table2* has been replaced by its actual record values, each one generating a query, the query cost would be:

$$\sum_{i=1}^{n2/d2} C(Q_i(t))$$

The smaller the value of *n2*, and the more selective the predicate on it (the higher the value of *d2*), then the lower the total query cost would be.

The tuple substitution procedure would chose *table1* to substitute for if *n1/d1* is less than *n2/d2*. It would chose *table2* if *n2/d2* < *n1/d1*.

To compare the cost with the original query, the same table example statistics will be used. Hence, let *table1* have 30 records and *table2* have 5000 records. Both variables can be reduced due to the extra predicates, which variable selection would perform in order to improve query processing in accordance with the cost minimisation algorithm of Ingres.

Applying the single variable predicates:
`(A.a = value_1) and (B.f  = value_2)`

`(A.a = value_1)` would leave *table1* with 24 tuples, instead of 30.

The predicate `(B.f  = value_2)` would leave *table2* with 2000 records (assuming that there are 2000 rows with `B.f = value_2).`

Hence, the ranges of variables have been reduced. *Table1* has been reduced from 30 to 24 tuples and *table2* has been reduced from 5000 to 2000 tuples.

The SI query requires a total of 24 + 500 = 524 tuple-substituted queries, compared to 530 for the original queries. 500 of the queries required to process the SI query are on a table (*table_temp*) of significantly reduced size - due to rules applied in *Q1*, which made it much smaller, prior to it being used by *Q2*. According to the cost estimate based on the number of pages in the queried table (*table_temp*), each of the 500 queries, produced for *Q2*, will have 40% of the cost of counterpart queries in the original (ignoring issues of data fragmentation) query.

Therefore, the greater the selectivity is, increased by SI introducing additional predicates, the fewer the number of single variable queries there will be. Assuming that each single variable query has the same cost due to being

processed in the same way by the query optimiser, then SI reduces the query processing cost by the same proportion that it increases the selectivity by.