Is Semantic Query Optimization Worthwhile?

A thesis
submitted in partial fulfilment
of the requirements for the degree
of

Doctor of Philosophy in Computer Science
at
The University of Waikato

by

Bryan H. Genet

The University of Waikato
December 2006

For Luca

Tu es mon Soleil et ma Lune.
Ce que je n'ai jamais auparavant donné,
À toi je le donne librement.

Abstract

The term "semantic query optimization" (SQO) denotes a methodology whereby queries against databases are optimized using *semantic information* about the database objects being queried. The result of semantically optimizing a query is another query which is syntactically different to the original, but *semantically equivalent* and which may be answered more efficiently than the original. SQO is distinctly different from the work performed by the conventional SQL optimizer. The SQL optimizer generates a set of logically equivalent alternative execution paths based ultimately on the rules of relational algebra. However, only a small proportion of the readily available semantic information is utilised by current SQL optimizers. Researchers in SQO agree that SQO can be very effective. However, after some twenty years of research into SQO, there is still no commercial implementation. In this thesis we argue that we need to quantify the conditions for which SQO is worthwhile. We investigate what these conditions are and apply this knowledge to relational database management systems (RDBMS) with static schemas and infrequently updated data. Any semantic query optimizer requires the ability to *reason* using the semantic information available, in order to draw conclusions which ultimately facilitate the recasting of the original query into a form which can be answered more efficiently. This reasoning engine is currently not part of any commercial RDBMS implementation. We show how a practical semantic query optimizer may be built utilising readily available semantic information, much of it already captured by meta-data typically stored in commercial RDBMS. We develop cost models which predict an upper bound to the amount of optimization one can expect when queries are pre-processed by a semantic optimizer. We present a series of empirical results to confirm the effectiveness or otherwise of various types of SQO and demonstrate the circumstances under which SQO can be effective.

Acknowledgements

It is just as well we cannot see into the future.

When I returned to the University of Waikato at the beginning of 2004, I was determined to complete my research into semantic query optimization, which I had only just begun at Victoria University of Wellington before succumbing to the siren call of IT contract work overseas. I brought with me nearly 15 years of experience in database technology and reassured myself I was well placed to undertake a piece of research with a high practical content. I was determined to write a thesis that would be comprehensible to industry practitioners, while breaking sufficient new ground academically to satisfy the demands of the university post-graduate environment.

The University of Waikato seemed the natural place for me to resume this research. I had been a student here in the 1970s, completing a first degree in Physics and was privileged to be lectured by the late Dr Roger Osbourne, Dr Crispen Gardener, the late Dr Dan Walls and many others who profoundly influenced the direction of my life and equipped me for a lifetime of learning. I returned to Waikato in the 1980s to complete another undergraduate major in Computer Science. The Department of Computer Science at that time was buoyant and thriving and I benefited from the expertise of Dr Olivier de Val, Mr Bill Rogers, Dr Ian Graham, Professor E.V. Krishnamurthy and others. It was the gentle eccentricity of Professor Krishnamurthy that first inclined me toward post-graduate research and I soon found myself at the University of Melbourne, immersed in Prolog and AI research.

Times and fashions change and Computer Science is as subject to this as anything else. I resumed my research into semantic query optimization and decided the focus of my research ought to be the deeply unfashionable area of relational database management systems (RDBMS). My decision was based on the observation that over 90% of real world industry data was contained, queried and manipulated by RDBMS and the question of utilising the semantics of this data to optimize queries had most definitely not been settled. I was of course unable to foresee the joys, and the sorrows, that would ensue from such a decision.

Arriving now at the cusp of another era in my life drawing to a close, I have many people whom I wish to thank for their help and encouragement over the last three years. Firstly I thank my Supervisor, Dr Sally Jo Cunningham, who was brave enough to take me on when I first arrived with my proposal for PhD research and whose on going support, constructive criticism and academic experience has enabled me to navigate the sometimes perilous course of this research. I am grateful to Dr Geoff Holmes for his ability to provide grounded, no nonsense advice with an uncanny timeliness and for his review of some of the material in this thesis. I thank Dr Gill Dobbie of Auckland University for her consistently positive, wise advice and for support that extends back to my time at Melbourne University. I am grateful to Dr Annika Hinze for her collaboration early in this research and for some later reviews. The financial support I have received via the Departmental scholarship and the Graduate Assistant scheme has been pivotal and I am grateful to have been judged worthy of that support.

Outside of the academic realm, I have been encouraged and sustained by the friendship of many of the post-graduate students in the Department. Sven Bittner and Doris Jung have been patient and interested fellow travellers. Phil Treweek has been far more than a Senior Tutor, skillfully providing pastoral care when it was required. I thank Dr David Streader for having enough life experience to allow me to be myself. I am grateful to Rob Akscyn for being able to speak with the authority of one who has actually been there and done it. My thanks to Dr Bernhard Pfahringer for appreciating my teaching and saying so.

I thank my parents and my wonderful adult children for their support and acceptance of the somewhat eccentric orbit of my life. Finally, I am deeply grateful to my dear friend Sue Jury, whose professionalism, insight and constant support has sustained me through this journey.

Table of Contents

List of Figures

List of Tables

Is Semantic Query Optimization Worthwhile?

A thesis
submitted in partial fulfilment
of the requirements for the degree
of

Doctor of Philosophy in Computer Science
at
The University of Waikato

by

Bryan H. Genet

The University of Waikato
December 2006

For Luca

Tu es mon Soleil et ma Lune.
Ce que je n’ai jamais auparavant donné,
À toi je le donne librement.

Abstract

The term "semantic query optimization" (SQO) denotes a methodology whereby queries against databases are optimized using *semantic information* about the database objects being queried. The result of semantically optimizing a query is another query which is syntactically different to the original, but *semantically equivalent* and which may be answered more efficiently than the original. SQO is distinctly different from the work performed by the conventional SQL optimizer. The SQL optimizer generates a set of logically equivalent alternative execution paths based ultimately on the rules of relational algebra. However, only a small proportion of the readily available semantic information is utilised by current SQL optimizers. Researchers in SQO agree that SQO can be very effective. However, after some twenty years of research into SQO, there is still no commercial implementation. In this thesis we argue that we need to quantify the conditions for which SQO is worthwhile. We investigate what these conditions are and apply this knowledge to relational database management systems (RDBMS) with static schemas and infrequently updated data. Any semantic query optimizer requires the ability to *reason* using the semantic information available, in order to draw conclusions which ultimately facilitate the recasting of the original query into a form which can be answered more efficiently. This reasoning engine is currently not part of any commercial RDBMS implementation. We show how a practical semantic query optimizer may be built utilising readily available semantic information, much of it already captured by meta-data typically stored in commercial RDBMS. We develop cost models which predict an upper bound to the amount of optimization one can expect when queries are pre-processed by a semantic optimizer. We present a series of empirical results to confirm the effectiveness or otherwise of various types of SQO and demonstrate the circumstances under which SQO can be effective.

Acknowledgements

It is just as well we cannot see into the future.

When I returned to the University of Waikato at the beginning of 2004, I was determined to complete my research into semantic query optimization, which I had only just begun at Victoria University of Wellington before succumbing to the siren call of IT contract work overseas. I brought with me nearly 15 years of experience in database technology and reassured myself I was well placed to undertake a piece of research with a high practical content. I was determined to write a thesis that would be comprehensible to industry practitioners, while breaking sufficient new ground academically to satisfy the demands of the university post-graduate environment.

The University of Waikato seemed the natural place for me to resume this research. I had been a student here in the 1970s, completing a first degree in Physics and was privileged to be lectured by the late Dr Roger Osbourne, Dr Crispen Gardener, the late Dr Dan Walls and many others who profoundly influenced the direction of my life and equipped me for a lifetime of learning. I returned to Waikato in the 1980s to complete another undergraduate major in Computer Science. The Department of Computer Science at that time was buoyant and thriving and I benefited from the expertise of Dr Olivier de Val, Mr Bill Rogers, Dr Ian Graham, Professor E.V. Krishnamurthy and others. It was the gentle eccentricity of Professor Krishnamurthy that first inclined me toward post-graduate research and I soon found myself at the University of Melbourne, immersed in Prolog and AI research.

Times and fashions change and Computer Science is as subject to this as anything else. I resumed my research into semantic query optimization and decided the focus of my research ought to be the deeply unfashionable area of relational database management systems (RDBMS). My decision was based on the observation that over 90% of real world industry data was contained, queried and manipulated by RDBMS and the question of utilising the semantics of this data to optimize queries had most definitely not been settled. I was of course unable to foresee the joys, and the sorrows, that would ensue from such a decision.

Arriving now at the cusp of another era in my life drawing to a close, I have many people whom I wish to thank for their help and encouragement over the last three years. Firstly I thank my Supervisor, Dr Sally Jo Cunningham, who was brave enough to take me on when I first arrived with my proposal for PhD research and whose on going support, constructive criticism and academic experience has enabled me to navigate the sometimes perilous course of this research. I am grateful to Dr Geoff Holmes for his ability to provide grounded, no nonsense advice with an uncanny timeliness and for his review of some of the material in this thesis. I thank Dr Gill Dobbie of Auckland University for her consistently positive, wise advice and for support that extends back to my time at Melbourne University. I am grateful to Dr Annika Hinze for her collaboration early in this research and for some later reviews. The financial support I have received via the Departmental scholarship and the Graduate Assistant scheme has been pivotal and I am grateful to have been judged worthy of that support.

Outside of the academic realm, I have been encouraged and sustained by the friendship of many of the post-graduate students in the Department. Sven Bittner and Doris Jung have been patient and interested fellow travellers. Phil Treweek has been far more than a Senior Tutor, skillfully providing pastoral care when it was required. I thank Dr David Streader for having enough life experience to allow me to be myself. I am grateful to Rob Akscyn for being able to speak with the authority of one who has actually been there and done it. My thanks to Dr Bernhard Pfahringer for appreciating my teaching and saying so.

I thank my parents and my wonderful adult children for their support and acceptance of the somewhat eccentric orbit of my life. Finally, I am deeply grateful to my dear friend Sue Jury, whose professionalism, insight and constant support has sustained me through this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preamble

The term "semantic query optimization" (SQO) denotes a methodology whereby queries against databases are optimized using *semantic information* about the database objects being queried (Date 2003*b*). Semantic information includes *schema meta-data* (such as the table and view definitions in a relational database), *domain knowledge* (such as might be held by human domain experts) as well as various *constraints* defined, stored and enforced by the database management system (DBMS). The result of semantically optimizing a query is another query which is syntactically different to the original, but *semantically equivalent* and which may be answered more efficiently than the original (Godfrey, Gryz & Minker 1996). The original query and the transformed query are said to be semantically equivalent if they produce the same answer, for a given database state (Siegel, Sciore & Salveter 1992).

Informally, when we semantically optimize a query, we rewrite that query using knowledge we have about the domain of interest, such that the rewritten query extracts the same answer from the database more quickly. It is important to distinguish the query rewrite we refer to here in the context of SQO from the work performed by the conventional SQL[1] optimizer. The SQL optimizer generates a set of logically equivalent alternative execution paths based ultimately on the rules of relational algebra (Waas & Galindo-Legaria 2000). However, in general *semantic* information is not utilised by current SQL optimizers (Genet & Dobbie 1998)[2].

Researchers in SQO agree that SQO can be very effective (Yoon, Henschen, Park & Makki 1999, Hsu & Knoblock 1994). However, after some twenty years of research into SQO, there is still no commercial implementation (Cheng, Gryz, Koo, Leung, Liu, Qian & Schiefer 1999, Godfrey, Gryz & Zuzarte 2001). In this thesis we argue that we need to quantify the conditions for which SQO is worthwhile. We investigate what these conditions are and apply this knowledge to relational database management systems (RDBMS) where tables are large enough to provoke significant disk activity and where standard column indexes are consulted.

We make few assumptions about the database schemas we investigate throughout this thesis, apart from the following:

[1]SQL: "Structured Query Language" is a high-level nonprocedural data language implemented in almost every commercial DBMS. The original authors were D.D. Chamberlin and R.F. Boyce at the IBM San Jose Laboratory in the 1970s (Chamberlin & Boyce 1974). E.F Codd described the relational model for databases while working for IBM in the 1970s. The commercial acceptance of SQL was precipitated by the formation of SQL Standards committees by the American National Standards Institute and the International Standards Organization in 1986 and 1987 (Date 2003*a*, Eisenberg & Melton 2000).

[2]There are some minor exceptions to this general observation. These are described in Section 2.6, page 37.

1. The relational tables which are the targets of our queries and joins are "large"; i.e., significant computational resources are required to satisfy the queries we pose. Our objective here is *realism*. We report results for tables that are realistically indexed, which run to 10^6 rows and which require a few gigabytes of disk storage to contain[3].

2. We focus on scenarios where *schemas* evolve only slowly or not at all. In particular, we assume that the various *schema constraints*, stored as part of the normal RDBMS meta-data, are constant. This is typically the case for *data warehouses*[4] (Hobbs et al. 2004).

3. We assume throughout this thesis that data in our target tables are updated only infrequently. We do not, for example, consider transactional environments where data is constantly changing. Infrequent data update is the norm for data warehouses where typically data is inserted in batch mode, additional data is added rather than existing data updated and the frequency of data refresh is slow (say, once every 24 hours or once per week) compared to the querying frequency (Lane & Schupmann 2002).

The first point above recognises that SQO is not costless. Indeed, we show unequivocally in this thesis that any implementation of semantic optimization must necessarily incur costs which quickly become comparable to or exceed the normal computational costs incurred by SQL query optimizers. It is unlikely therefore that queries which currently require scant resources to answer will benefit from SQO.

The second and third points above underline the common sense notion that SQO can never work unless relevant semantic information is first collected and made available in a form which can be utilised by the semantic optimizer. We show in this thesis that existing schema constraints are a rich source of semantic information which are utilised for data insert or modification, but are currently largely ignored by existing SQL optimizers for querying. In order for this information to be utilised, we assume that it is constant or changes so infrequently as to make the cost of updating it insignificant. Similarly, analysis of the data itself contained within the relational tables can yield valuable semantic rules which can be used to semantically optimize queries (Chen 1996). We assume that such rules are not constantly invalidated by regular data updates such as might occur in a transaction processing environment.

[3]The term "realistically", in this context, denotes indexing strategies which reflect current industry best practice. Specifically, columns which are frequently cited in the restriction clauses of SQL queries and which are sufficiently selective, are candidates for indexing with standard B-tree indexes (Cyran & Lane 2003, Date 2003*b*, Burleson 1994). Indexing *per se* is beyond the scope of this thesis.

[4]More detailed consideration of data warehouses per se is beyond the scope of this thesis. See for example (Hobbs, Hillson, Lawande & Smith 2004, Lane & Schupmann 2002).

Any semantic query optimizer requires the ability to *reason* using the semantic information available, in order to draw conclusions which ultimately facilitate the recasting of the original query into a form which can be answered more efficiently. This reasoning engine is currently not part of any commercial RDBMS implementation. We show that an effective reasoning engine must not only have a sound theoretical base, but be able to utilise semantic rules which are derived from a variety of sources including schema meta-data, domain knowledge, schema constraints and correlations that may exist in tabular data. The basis of our reasoning engine is an *interval algebra* which we show can readily be used to deduce the very conclusions required by an effective semantic query optimizer.

The main contributions of this thesis are as follows:

- We present a thorough analysis of research in SQO. We introduce definitions that clarify and simplify the terminology used by other researchers. In addition, further definitions are introduced that enable a more detailed discussion.

- We develop a sound theoretical base for our study using an *interval algebra*, which we show may be built using only a small number of well understood and researched axioms. We extend the interval algebra by defining an *interval list* data structure which we subsequently utilise as the basic data structure of our implementation. To our knowledge, this is the first report of an interval algebra used in the way we describe and generalised to operate with any data type that has a deterministic *total ordering*.

- We show how a practical semantic query optimizer may be built utilising readily available semantic information, much of it already captured by meta-data typically stored in commercial RDBMS. We describe how SQO may proceed as a series of pre-processing steps which may be switched in and out as changing database conditions make different forms of SQO worthwhile. While other researchers have suggested the basic techniques we describe, we focus on the fact that certain types of SQO, such as the detection of unsatisfiable queries, are likely to be worthwhile *given a particular query profile*. We describe an extension to the detection of unsatisfiable queries which enables "data holes" to be discovered separately across all relevant dimensions (i.e., across all table columns that are *actually* cited in query restrictions) and incorporated incrementally into the semantic information utilised by the semantic optimizer with little or no impact on database usability. In addition, we develop a cost model which accurately predicts the amount of optimization we can expect and which sets a clear upper bound to this optimization. To our knowledge, this is the first report to explicitly highlight an inherent limitation on the effectiveness of detecting unsatisfiable queries and joins.

- We describe an empirical methodology which overcomes problems of repeatability and consistency which typically arise in experiments with RDBMS where automatic maintenance processes may be invoked outside of the control of the experimenter and where large query and data caches are available. We do not report results for individual queries but instead report statistical *averages* that arise from large batches of similar queries. Our results therefore inform us as to what we can expect from whole classes of queries rather than individual queries specific to particular databases.

- We present a series of empirical results arising from experiments to confirm the effectiveness or otherwise of various types of SQO. Our experiments are performed with tables which realistically reflect the conditions likely to be encountered in schemas with table objects large enough to provoke significant disk activity and where standard indexes are consulted. Crucially, we report results for tables that are realistically indexed. To our knowledge, this is the first report of empirical results for queries and equi-joins against tables that are indexed in this way and where the results are a statistical average for batches of many similar queries.

- We describe several important extensions which utilise our interval algebra. Firstly, we show how our interval algebra can be used to implement a novel type of *interval arithmetic.* Our interval arithmetic is more general than traditional implementations in that we allow both inclusive and exclusive upper and lower bounds for the numeric intervals. Furthermore, we show how the subtly different semantics of our implementation elegantly capture notions such as *plus and minus infinity* while allowing arithmetic calculation to proceed across a greater set of cases than allowed for by traditional interval arithmetic. Secondly, we show how our interval algebra subsumes the temporal algebra of Allen (Allen 1983) and how the *13 Allen interval relations* can be meaningfully extended.

1.2 Summary of Chapter Contents

We now summarise the content of the succeeding chapters.

- In Chapter 2 we present a thorough analysis of research in SQO. We first consider the discovery of semantic information and semantic rules and describe how other researchers have classified semantic rules according to the rules' reliability. This is followed by a precise definition of SQO itself. We then describe the main types of semantic query optimization. The inherent limitations of current SQL query optimizers are described. We conclude the chapter with a brief summary of a subset of SQO techniques currently implemented in some commercial RDBMS.

- In Chapter 3 we develop a sound theoretical base for our study using an *interval algebra* which we show may be built using only a small number of well understood and well researched axioms. We proceed to develop an *interval* definition which we show is equivalent to a sentence in first order predicate calculus. We give definitions for three basic operations on intervals which are forms of *conjunction*, *disjunction* and *negation*. We extend the interval algebra by defining an *interval list* data structure which we subsequently utilise as the basic data structure of our implementation. We show our interval list is equivalent to a *disjunction of disjoint intervals* and that as a consequence we may develop sound definitions for conjunction, disjunction and negation that extend to interval lists.

- In Chapter 4 we show how a practical semantic query optimizer may be built utilising readily available semantic information. We begin by highlighting an intrinsic limiting factor in semantic optimization. We then explain how conventional database constraints may be utilised as the initial step in the harvesting of relevant semantic rules. We then describe how SQO may proceed as a series of pre-processing steps which can be switched in and out as changing database conditions make different forms of SQO worthwhile. We conclude the chapter with a detailed description of how we utilise conditional semantic rules and how these rules are triggered using the *subsumption rule* which we incorporate into our *reasoning engine*(RE).

- In Chapter 5 we firstly justify our choice of the Oracle™RDBMS[5] for our experiments. We then explain the difficulties of obtaining repeatable, consistent results with RDBMS which have automatic maintenance processes executing beyond the control of the experimenter and which have large query and data caches available. We explain our choice of cost metrics which we use to measure the computational cost of a query and justify the use of a combined metric which averages the computational costs incurred from the perspective of *elapsed time*, *disk i/o* and *CPU time* respectively. We describe a qualitative classification of query complexity or difficulty which we subsequently use to characterise the content of the query batches we submit to the database in our experiments. We conclude the chapter by developing a comprehensive cost model which we use to accurately predict the amount of optimization we can expect from detecting unsatisfiable queries.

[5] *Oracle* is the trademark of *Oracle Corporation* (see `http://www.oracle.com`). We refer to Oracle's RDBMS throughout this thesis simply as "Oracle".

- In Chapter 6 we present our main empirical results. We describe our hypotheses, experimental methodology and conclusions reached from our experiments. We focus firstly on the amount of optimization we can expect from a given query profile and the success of our cost model in predicting an upper bound to this optimization. We show unequivocally that SQO is not costless and that pre-processing queries in the manner we describe rapidly incurs computational costs that are comparable to and exceed the costs of normal SQL optimization. We demonstrate two simple but effective query rewrite techniques. We conclude the chapter with results which demonstrate the effectiveness of:

  – introducing extra restrictions to the query which do not change the query's semantics but which might be expected to increase its speed;

  – removing restrictions from the query which do not change the query's semantics but which might be expected to increase its speed.

- In Chapter 7 we describe several important extensions which utilise our interval algebra. Firstly, we show how our interval algebra can be used to implement a novel type of *interval arithmetic*. Secondly, we show how our interval algebra subsumes the temporal algebra of Allen (Allen 1983) and how the *13 Allen interval relations* can be meaningfully extended.

- In Chapter 8 we conclude the thesis by first summarising the main research contribution, followed by a description of future research directions arising from this current work.

1.3 Motivating Background

The remainder of this chapter is to provide motivation for the reader by informally describing some essential background information.

- We briefly describe in Section 1.3.1 how an SQL query is parsed and executed with the assistance of current SQL language optimizers.

- We then describe some inherent limitations of current SQL query optimizers in Section 1.3.2.

- We close the chapter with a simple motivating example in Section 1.3.3.

1.3.1 Typical SQL Query Processing

We now briefly describe how an SQL query is parsed and executed with the assistance of current SQL language optimizers[6]. Consider a database table `CUSTOMER` which stores information about customers including a unique `ID` (the primary key of the table) along with the customer's `NAME`, `ADDRESS` and `TELEPHONE`. A database user queries the database by submitting query text, typically using the query language SQL. Two queries are submitted:

- `select * from CUSTOMER where ID = 999;`
- `select * from CUSTOMER where NAME = 'SMITH';`

1. The SQL text is first *parsed*. "Parsing" in this context means first verifying it to be a syntactically valid statement and then performing data dictionary lookups to check table and column references are valid.

2. The parsed SQL query is then passed to the *optimizer*. SQL requires an optimizer because it is a declarative language (Date 2003*b*). The task of the SQL optimizer is to determine an actual execution path which is logically equivalent to the parsed SQL query. Typically there will be many choices of possible path with the result that determining the *optimal* path rapidly becomes exponentially difficult. All SQL optimizers employ heuristics to overcome this problem (Sciore & Siegel 1990, Shenoy & Ozsoyoglu 1989, King 1981) returning the execution plan estimated to return the query answer most efficiently.

3. The query is then *executed*; i.e., data is fetched and returned to the user by following the execution plan that was returned by the optimizer in the previous step.

The most influential factor on the choice of actual execution plan chosen by the optimizer is the presence or absence of an *index* on the columns that occur in the query restrictions. For example, in the first of the above queries, an index on column `ID` might be consulted by the optimizer while in the second query an index on column `NAME` might be consulted. Conversely, if no index exists on the column cited in the query restriction, the entire table must be loaded and each row checked individually to determine if it satisfies the restriction. For example, in the second

[6]We do not provide a rigorous description. SQL query answering and the detailed operation of the language optimizer are beyond the scope of this thesis. We assume a very basic reader knowledge of relational databases (RDB) such as data storage in tabular format, the existence of the *data dictionary* and the notions of *primary key* and *index*. Much of this background material is inspired by introductory material in (Date 1995)

example above, if no index exists on column `NAME`, this would provoke the loading of the entire table into memory to enable each row to be checked for the name `'SMITH'`.

A useful index then will be one that points *directly* to the location on disk of the requested data items, allowing the data to be returned quickly with a minimum of extra processing. The usefulness of an index is typically characterised by its *selectivity*, which simply expresses the uniqueness of the values stored in the indexed column. Therefore, all primary key columns are said to be 100% selective because each primary key value is unique. Suppose the `CUSTOMER` table in the above example has 1,000,000 rows but there are just 1,000 unique names in the `NAME` column. Then column `NAME` has a selectivity of just 0.1%. In other words, the first query above will return at most one row while the second query might return of the order of 1,000 rows.

One of the important tasks of the optimizer is to estimate the cardinality of the result set returned by successive stages of the execution plan. The optimizer will generally choose a plan that minimises the cardinality of the result set as early as possible in the computation. If the selectivity of an index is too low, the optimizer may ignore it because the expense of loading into memory index values followed by the data rows pointed to by the index, exceeds the expense of simply loading the entire table into memory. For example, in the second query above, it is unlikely the optimizer would choose to consult the index on column `NAME` with a selectivity of just 0.1%.

1.3.2 Limitations of Current SQL Optimizers

We now describe some inherent limitations of current SQL query optimizers. These limitations arise not as a result of any intrinsic lack or inefficiency in the available commercial optimizers, but because their design is limited to producing execution paths that are logically equivalent to the original SQL query text, utilising the rules of the relational algebra (Date 2003*b*). In general, they do not utilise semantic information and are unaware of, for example, all but a small proportion of readily available schema constraints encoded within the database itself (Genet & Dobbie 1998). While SQL optimizers apply the rules of relational algebra, they lack a *semantic* reasoning engine (Hsu & Knoblock 2000, Godfrey et al. 1996) and are therefore typically unable to use semantic information to influence the choice of a suitable execution path.

- They are unable to detect logical inconsistencies in query text. For example, consider the following query on a RDB table `CUSTOMER` containing column `CITY`:

```
select * from CUSTOMER where CITY > CITY;
```

Current SQL optimizers will submit such a query to the database unmodified.

- They are unable to detect inconsistencies between the table DDL and the query text. For example, suppose the same table `CUSTOMER` includes a column `STATUS` which the table's definition declares to be "`char(1)`" indicating the column stores a single character:

  ```
  create table CUSTOMER(
    ID      number(8),
    NAME    varchar2(30),
    ADDRESS varchar2(30),
    STATUS  char(1),
    .
    .
    .
  );
  ```

  Now consider a query such as:

  ```
  select * from CUSTOMER where STATUS = 'ok';
  ```

  The length of the literal "`ok`" is two, so this query is unsatisfiable. Nevertheless, current SQL optimizers will submit such a query to the database unmodified.

- They are unable to resolve schema constraints with query restrictions. For example, consider a query on the same table `CUSTOMER` in the presence of the schema constraint:

  ```
  check CITY in ('London','Paris','New York');
  ```

  Now consider a query such as:

  ```
  select * from CUSTOMER where CITY = 'Auckland';
  ```

  Again, this query will be submitted to the database unmodified.

SQL language optimizers are generally classified as *rule based* or *cost based* (Babcock & Chaudhuri 2005, Warshaw & Miranker 1999, Haas, Carey, Livny & Shukla 1997, Cherniack & Zdonik 1996). With rule based optimization, the optimizer chooses an execution plan based on the access paths available and the ranks of these access paths. The ranking of the access paths is heuristic. If there is more than one way to execute a SQL statement, then the rule based optimizer always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank (Cherniack & Zdonik 1998, Luscher & Green 2002).

Both rule and cost based optimizers ultimately rely on the relational algebra to produce an execution plan, but the cost based optimizer uses additional heuristics. With cost based optimization, the optimizer uses height-balanced histograms to estimate the distribution of column data and hence the cardinality of the result set. For example, consider a query which is restricted on an indexed column. The cost based optimizer may infer the cardinality of the result set is too high to justify an index scan, ordering a full table scan despite the presence of a relevant index (Waas & Galindo-Legaria 2000, Chan 2005*d*)[7].

1.3.3 Motivating Example

We conclude this introduction with a simple motivating example which captures how the technique of semantic query optimization achieves its speed-up of query answering.

Example 1.3.1. *The Human Resources Manager of a global IT company poses the following query to the company database:*

"Give me all the information about employees with Computer Science degrees who earn less than $50,000 per year."

Being a global IT company, its employee database is large and the Manager observes the results of the query take some minutes to appear. The query in fact returns no rows. This is because all Computer Science graduates in this company receive salaries in excess of $50,000. A closer inspection of the employee data would have yielded the simple semantic rule:

"If an employee has a Computer Science degree then their salary is greater than $50,000."

This is precisely the type of rule utilised by a semantic query optimizer. Given the original query, the semantic optimizer would deduce *that this query cannot return any rows, thus obviating the need to even submit the query to the database. In the presence of such a semantic optimizer, the answer to the Manager's query would appear instantaneously.*

[7]Current commercial RDBMS generally recommend cost based optimization. A detailed study of the SQL optimizer is beyond the scope of this thesis. We simply assume its presence throughout.

Chapter 2

Background Research

2.1 Introduction

After two decades of research into semantic query optimization (SQO), there is still no commercial implementation and SQO is neither mainstream nor widely employed (Cheng et al. 1999, Godfrey et al. 2001). Nevertheless there is clear agreement in the literature that SQO is able to speed up certain types of database queries (Yoon et al. 1999, Date 2003*b*, Hsu & Knoblock 1994).

The main emphasis of SQO research has been deductive databases. (For example, see (Cheng et al. 1999, Godfrey et al. 2001)). However in this thesis, we specifically address the role of SQO in relational databases (RDB). We focus on the querying of large table objects such as those typically found in data warehouses.

It is important to distinguish SQO from the conventional query optimization performed by all commercial relational database management systems (RDBMS)[1]. When we speak of *conventional* SQL query optimization in the context of RDBMS, we are specifically referring to the task performed by the *SQL query optimizer.* All RDBMS include such SQL optimizers. The performance of the SQL optimizer is of central importance to a RDBMS and is one of the main ways that the various commercial RDBMS distinguish themselves from each other. The function of the SQL query optimizer is beyond the scope of this thesis and we simply assume its presence throughout.

The remainder of this chapter is organised as follows.

- We describe related research in the field of SQO. Our description is built around the definitions of some important terms. We begin by considering preliminary definitions used by other researchers in the field of SQO (Section 2.2).
- We then focus on the use of semantic information and semantic rules (Section 2.3) before defining the principle term "semantic query optimization" itself (Section 2.4).
- We set out the main types of SQO as they have been classified by other researchers (Section 2.5).
- We reiterate some inherent limitations of current SQL query optimizers which are shared by all commercial RDBMS (Section 1.3.2).
- We consider SQO in the context of commercial RDBMS and report some of the reasons suggested by other researchers as to why SQO is not routinely employed.

[1]Well known commercial RDBMS include *Oracle™* (see `http://www.oracle.com`), *MS SQL Server™* (see `http://www.microsoft.com/sql`), *DB2™* (see `http://www.ibm.com`), *Sybase™* (see `http://www.ianywhere.com`), *Postgres* (see `http://www.postgresql.org`).

- We give a summary of a small set of semantic optimizations which are implemented in some commercial RDBMS (Section 2.6).
- We conclude by listing the main contributions of the chapter (Section 2.7).

2.2 Preliminary Definitions

We now describe related research in the field of SQO by considering the definitions of some important terms used in the research literature. Our goal is to clarify and to some extent simplify some of this terminology. We will define the central term *semantic query optimization* in due course, but in order to do so we first define some preliminary terms.

The aim of SQO is to use semantic knowledge about the database domain to transform a query into one which is syntactically different but *semantically equivalent* and which can be executed more efficiently [2] (Godfrey et al. 1996, Siegel et al. 1992, Chakravarthy, Grant & Minker 1990, King 1981, Hammer & Zdonik 1980).

The queries in question cannot meaningfully be considered equivalent *per se* but must be semantically equivalent *with respect to some specific set of schema semantics*. So it is more correct to say that *semantic equivalence* means the transformed query always produces the same answer as the original query for any database state *satisfying a given set of schema constraints* (Siegel et al. 1992). In this thesis, we assume a database schema is available and we subsequently utilise the schema as the first step in the construction of a practical semantic query optimizer. We focus explicitly on schema constraints beginning at Definition 2.3.1 on page 19.

For RDBMS, the semantic equivalence of two queries means, literally, that the tuples returned by the two queries are exactly the same for a given database state, although the result set is not necessarily presented in the same order. Since the contents of a database will typically change over time, whenever we speak of the *answer* to a query we are referring to the result set (i.e., the tuples) returned at a particular point in time for a particular database state. If new data is inserted or existing data updated or deleted, then the database state has changed and the same query may return a different answer set.

Definition 2.2.1. ***Semantic equivalence****: Two database queries are semantically equivalent if they return the same answer, for a given database state.*

The notion of *query rewrite* is intrinsic to SQO and refers to the transformation or recasting of the query to a different textual expression (Aberer & Fischer

[2]The term *syntactically different* here refers to the actual textual expression of the query statement. We do not mean the abstract syntax of the query language SQL. See Definition 2.2.5 on page 17 for a precise definition of *query efficiency*.

1995, King 1981, Hammer & Zdonik 1980). It is important to distinguish the query rewrite we refer to here in the context of SQO from the work performed by the conventional SQL optimizer. The SQL optimizer generates a set of logically equivalent alternative execution paths based ultimately on the rules of relational algebra. That is, SQL optimizers perform *syntactic* optimization (Date 2003*b*) via a set of rewrite rules based on the notion of equality, as it is defined in the relational algebra. In general, *semantic* information is not utilised by current SQL optimizers (Genet & Dobbie 1998).

Definition 2.2.2. ***Query rewrite****: This is the process of rewriting a database query into a semantically equivalent query with a different syntax .*

All SQL language interpreters require an *optimizer* because SQL is a declarative language (Date 2003*b*). The task of the SQL optimizer is to determine an actual execution path which is logically equivalent to the original SQL query text. Typically there will be many choices of possible path with the result that determining the *optimal* path rapidly becomes exponentially explosive. Thus finding an optimal path in these circumstances is classified as NP-hard (Sun & Yu 1994). All SQL optimizers employ heuristics to overcome this problem (Sciore & Siegel 1990, Shenoy & Ozsoyoglu 1989, King 1981) with the desirable outcome being a *near-optimal* execution path.

An important feature of the SQL optimizer is that it calculates a dimensionless metric which is a measure of how costly the query is expected to be, without actually executing the query. This metric is readily available in all commercial RDBMS. This makes it straightforward to *estimate* the relative computational cost of equivalent execution paths, without actually executing the query many times and comparing average query times.

Definition 2.2.3. ***SQL optimizer****: This is the engine that takes as its input the SQL query text and outputs the specific execution path to be followed in order to compute the query answer.*

The notion of *query cost* is used very generally by researchers, hence the generic definition below (Definition 2.2.4). Various writers emphasize the amount of disk activity required to answer a query (Siegel et al. 1992), the total time required to answer the query (Zhu 1992) and delays due to communication costs (Jarke & Koch 1984). In this thesis, we focus on the *average* cost of submitting a query as part of a large batch of similar queries. We use a number of different metrics to evaluate the computational cost of a query, including total CPU time, total number of data blocks physically read from disk and the total elapsed time. This is described in detail in Chapter 5 (page 131).

Definition 2.2.4. ***Query cost***: *The cost of a query is the expenditure of computing resources required to answer that query. Expenditure is measured with a set of nominated metrics which typically include some permutation of total CPU time, total number of data block reads and total elapsed time.*

Intuitively, a more efficient query is one that on average requires fewer resources to execute (e.g. fewer disk reads) and therefore is executed more quickly (Godfrey & Gryz 1996), given a particular set of computer resources. One can only meaningfully compare the efficiencies of two queries if they are semantically equivalent. Consider two semantically equivalent queries Q_1 and Q_2. Query Q_1 is *more efficient* than query Q_2 if on average it is less costly, as indicated by some specific metric such as total elapsed time.

Definition 2.2.5. ***Query efficiency***: *This is the relative cost of semantically equivalent queries.*

Intuitively, an *unsatisfiable query* is posed against a particular schema and produces an empty result set, for a given database state. This empty result set might occur because of a *logical* contradiction embodied by the query, or simply because for the given database state, there is no data which satisfies the query conditions.

In this thesis we consider two sources of unsatisfiability in queries which result from a logical contradiction embodied by the query itself.

1. The unsatisfiability arises from a logical contradiction within the query text itself and independent of schema semantics. The following example illustrates how such a logical contradiction can arise.

   Example 2.2.1. *Consider a table* `TAB` *against which the following query is posed:*

   ```
   select *
   from   TAB
   where  1 = 2;
   ```

   This query will return an empty result set independent of any schema semantics or the contents of table `TAB` *simply because the restriction itself is arithmetically unsatisfiable.*

   It is perhaps surprising that such a simple scenario is not currently detected by at least one commercial RDBMS, in the sense that such a query will be submitted to the database in the normal manner. Why this is the case is described in detail in Section 1.3.2 (page 9).

2. The unsatisfiability arises from a logical contradiction between the query text and the schema semantics. The following example illustrates how such a logical contradiction can arise.

   Example 2.2.2. *In RDBMS, tables are defined using the database data definition language (DDL). Conventionally, numeric columns are declared to restrict the precision of the number, for example, to a designated number of decimal places. Consider a table* `TAB` *which we have defined by the following DDL:*

   ```
   create table TAB(
     COL1 number(1),
     COL2 varchar2(10),
     COL3 date
   );
   ```

   Column `COL1` *is constrained to be a single digit integer. Now consider the following query posed against table* `TAB`*:*

   ```
   select * from TAB where COL1 = 27;
   ```

   Clearly such a query will return no rows. It is unsatisfiable. This might have been deduced by simply considering the table's DDL.

The empty result set might occur because there is no data that satisfies the query conditions, for a given database state.

Example 2.2.3. *Consider a database schema populated with employee data. At a particular point in time, it happens that only data for female employees are found in the database. So any queries which ask for data on male employees will return empty result sets. For this database state, these queries are unsatisfiable.*

Detection of unsatisfiable queries is identified as pivotal by all researchers into SQO (Yoon et al. 1999, Genet & Dobbie 1998, Zhang & Ozsoyoglu 1997, Hsu & Knoblock 1996, Godfrey & Gryz 1996, Illarramendi, Blanco & Goni 1994). This is because, if we can deduce *a priori* that a query will produce an empty result set, it need not be posed to the database at all, resulting in a 100% saving (neglecting the cost of detecting the unsatisfiability). Unsatisfiable queries are considered in detail in Section 2.5.1 (page 31).

Definition 2.2.6. ***Unsatisfiable Query****: An unsatisfiable query is one which, for a given schema and database state, cannot return any rows.*

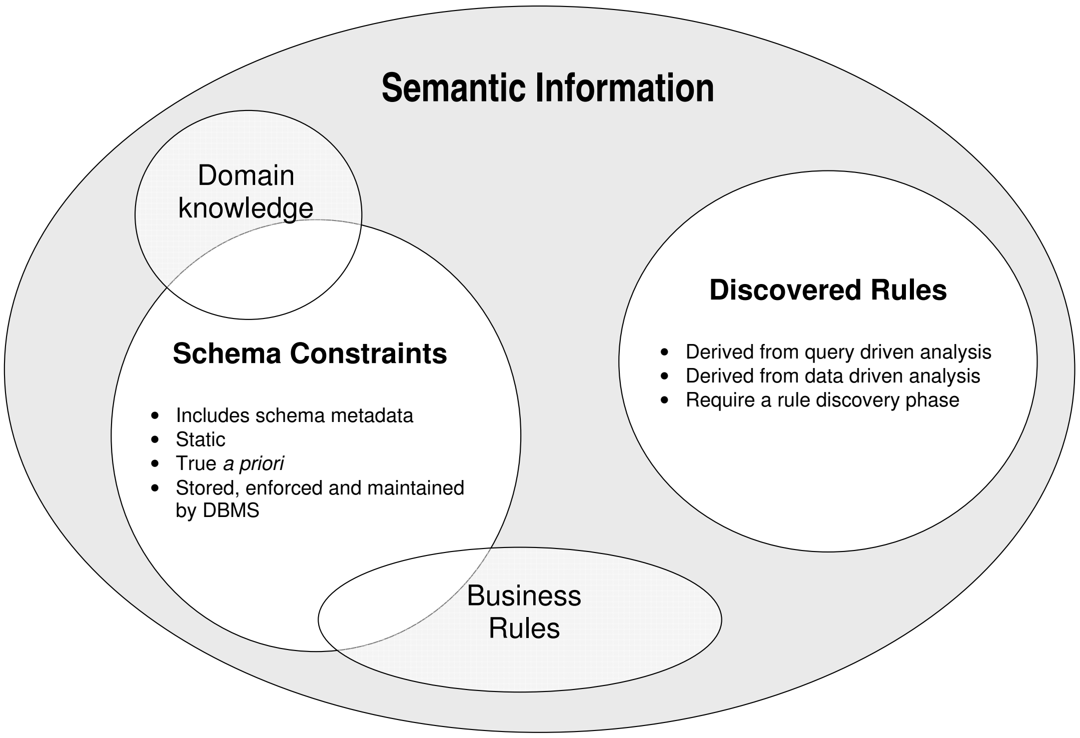

Semantic Information
Domain knowledge
Schema Constraints
• Includes schema metadata
• Static
• True *a priori*
• Stored, enforced and maintained by DBMS
Discovered Rules
• Derived from query driven analysis
• Derived from data driven analysis
• Require a rule discovery phase
Business Rules

Figure 2.1: **Sources of semantic information**: Semantic information can be drawn from a number of different sources including *schema constraints* and *discovered rules.* Schema constraints originate from human practitioners, while discovered rules are typically found by the execution of software.

2.3 Semantic Information and Semantic Rules

We now consider definitions arising from the notion of *semantic information.* We begin with a very general definition which we refine as we consider different sources of semantic information. Refer to Figure 2.1 on page 19. Semantic information includes schema meta-data (such as the table and view definitions in a relational database), domain knowledge (such as might be held by *domain experts*) as well as various constraints defined, stored and enforced by the database management system (DBMS). A prerequisite for query rewriting (Definition 2.2.2, page 16) is obtaining *valid* semantic information; i.e., semantic information which is true of the target database at this particular point in time.

Definition 2.3.1. ***Semantic information****: This is any logical statement or data which describes or constrains the data currently stored in the database and the data that may be stored in the database.*

The notion of *semantic rule* is intrinsic to SQO. The nuance of the term *rule* in this context is that the semantic information is captured by a formal logical sentence. Such logical sentences may be utilised by a reasoning system. In contrast, the terms *business rules* and *domain knowledge* do not necessarily refer to formal logical sentences and may include the informal knowledge of *domain experts.* Such knowledge may be difficult or impossible to express in first order predicate calculus (Godfrey et al. 1996).

Definition 2.3.2. ***Semantic rule****: A semantic rule is a sentence in first order predicate calculus which expresses semantic information.*

In this chapter we specifically identify a number of important sources of semantic rules. We classify these semantic rules firstly by observing whether or not they may be formally encoded within the database as *constraints*.

Definition 2.3.3. ***Schema constraint****: A schema constraint is a rule which is stored, maintained and enforced by the DBMS and which constrains the legal values that may be stored in the database.*

The critical role of schema constraints as a rich and stable source of semantic information is considered by (Yu & Sun 1989) and (Godfrey et al. 2001) in the context of knowledge discovery and deductive databases respectively. In this thesis we consider schema constraints in the context of RDBMS and specifically consider how they may be utilised for the purpose of SQO. This is considered in detail in Chapter 4 (page 95). For the time being we list the types of schema constraints found in RDBMS.

- *Schema meta-data*: The definitions of the various schema objects such as table definitions, view definitions and index definitions. It is helpful to think of the schema meta-data as the definitions created by the RDBMS *data definition language* (DDL).

- *Integrity constraints*: These are meta-data which define rules which constrain data at insert and modification. All commercial RDBMS allow the following constraints to be defined: *not null, primary key, unique key, foreign key, check.* Their purpose is to enforce relationships between data thus maintaining data integrity. The use of integrity constraints for SQO is considered in detail in Chapter 4 (page 95).

- *User defined constraints*: All commercial RDBMS allow constraints to be defined manually, typically to enforce complex business rules that would be difficult or impossible to implement using the built-in integrity constraints listed above. Such rules may be implemented as *triggers* which fire when certain conditions are met, for example, when new data is inserted[3]. User defined constraints are often built using a procedural language which may be an intrinsic part of the RDBMS. For example, the Oracle RDBMS includes

[3]Further discussion of triggers is beyond the scope of this thesis.

the procedural language *PL/SQL* . Alternatively, complex data relationships may be implemented, using a procedural language such as *Java* in a layer outside the actual database. This is the case when applications are built using a *three tier design methodology* (Doke, Satzinger, Williams & Douglas 2003) where the database is viewed simply as a persistence mechanism residing at the bottom of three tiers and all business rules are enforced procedurally in the middle tier[4].

Various researchers have identified other sources of semantic information (which may or may not be expressible in the language of the particular DBMS) such as *specialised domain knowledge* held by domain experts (Godfrey et al. 2001) and *application business rules* (Godfrey et al. 1996, Genet & Dobbie 1998, Date 2003*b*, Shekhar, Hamidzadeh, Kohli & Coyle 1993, Shenoy & Ozsoyoglu 1987).

We make the important observation that schema constraints are intended to constrain data only at insert or modification time and are not utilised at query time. Furthermore, while schema constraints remain valid statements about the domain of interest, this is no guarantee that the actual data stored in the database conforms to these rules. This contradictory situation arises with great regularity in commercial DBMS because constraints are often relaxed during data insert or modification. This is considered in more detail in Chapter 4 (page 95) where we describe the design of a practical semantic query optimizer.

Some researchers classify rules considered for semantic optimization as *static* or *dynamic* (Illarramendi et al. 1994, Chakravarthy et al. 1990). Included in static constraints are such rules that do not change or evolve over time as the state of the database changes. Therefore schema constraints, as we have defined them in Definition 2.3.3 (page 20), are static.

2.3.1 Rule Discovery

Some authors specifically concentrate on rules that are derived from the database (Chen 1996, Siegel et al. 1992). These studies use a variety of techniques to detect correlations in data which are then used to formulate rules. For example, in (Agrawal, Imieliński & Swami 1993) the authors present an algorithm for identifying correlations in sales data between sets of Boolean attributes. Techniques for discovering rules that exist between quantitative and categorical data in relational tables are described by (Srikant & Agrawal 1996). Mining association rules over data which specifically represent *intervals* is described by (Miller & Yang 1997). Rules discovered from the analysis of data are considered static until data updates invali-

[4]The top tier is typically the *graphical user interface* (GUI). Further discussion of three tier design is beyond the scope of this thesis.

date that assumption (Gryz, Schiefer, Zheng & Zuzarte 2001, Siegel et al. 1992).

Definition 2.3.4. ***Rule discovery****: Rule discovery is the search for patterns, regularities and correlations in the target database.*

Our definition follows (Gryz et al. 2001, Hsu & Knoblock 1998, Shekhar et al. 1993, Siegel et al. 1992), but for clarity specifically refers to the uncovering of semantic information which is *additional* to the schema constraints described above in Definition 2.3.3 (page 20). In (Shekhar et al. 1993), the authors use the term "rule discovery *phase*" to emphasize the uncovering of previously unknown information.

Discovered rules are typically of the form:

$$P_i \Rightarrow Q$$

where P_i is some finite conjunction of conditions on the tuples of a relational table and Q is a consequent condition (Miller & Yang 1997, Piatetsky-Shapiro 1991).

Example 2.3.1. *A rule discovery exercise on table `EMPLOYEE` discovers that all Managers over the age of 40 have salaries of at least \$150K. This information is formulated into the following rule:*

$$(POSITION = \text{'}MANAGER\text{'}) \ and \ (AGE > 40) \Rightarrow (SALARY \geq 150000)$$

When discovered rules are static, they need only be compiled once (Chakravarthy et al. 1990). This is an important consideration whenever significant computational resources must be expended to discover relevant rules. Static rules may be discovered "off line" so there is effectively no impact on database usability. Data warehouses are an important example where the database state typically evolves slowly enough for the computational expense of rule discovery to be worthwhile (Albrecht, Hümmer, Lehner & Schlesinger 2000).

Example 2.3.2. *A data warehouse comprising banking transactions is the subject of a rule discovery exercise. Any rules generated by the analysis of historic transactional data are likely to be static because the data is unlikely to ever be updated. Similarly, other parts of the data warehouse, such as customer details, are likely to evolve only slowly such that any discovered rules can be assumed to be valid for long enough to make worthwhile the computational expense of discovering such rules.*

However, *dynamic* rules may change as the database state changes. In particular, dynamic rules may be rendered invalid by updates to the database. When dynamic rules are used, the cost of checking to see if the rules are still valid after a database update must be taken into account. Therefore the use of dynamic rules may incur performance penalties at run time. To address this problem, in (Gryz et al. 2001) the authors suggest rule maintenance may be *deferred* after an update is found to be inconsistent with a rule. In this case, the rule is tagged as invalid and not utilised until it can be modified (or removed) to reflect the new semantics, at a time of low database activity. We consider the question of revalidating temporary or dynamic rules as quite separate from the central issue of SQO. Our experiments with discovered rules, which we describe in detail in Chapter 6 (page 153), make the assumption that the database state evolves slowly enough to make the rule discovery exercise worthwhile.

Many semantic rules discovered in a mechanical knowledge discovery process may be of no practical value to the semantic query optimizer (Godfrey et al. 2001, Cheng et al. 1999, Aberer & Fischer 1995). For example, no matter how strong the relationship captured by a discovered rule, it will nevertheless have no impact whatsoever if the rule is never actually invoked. In this sense, such a rule is not *relevant.* The following example clarifies the notion of rule relevance.

Example 2.3.3. *A knowledge discovery exercise identifies a strong correlation between an employee's bank, the make of their car and the gender of their manager. Such a correlation might be uncovered by a mechanical analysis which carries out an exhaustive search for such relationships. Yet this semantic knowledge, although valid, is of little practical value. A cursory examination of the actual queries made against the database reveals that queries incorporating the three attributes: bank, make of car and gender of manager are never actually submitted. So the (possibly expensive) knowledge discovery exercise is of no value.*

Definition 2.3.5. ***Rule relevance****: A semantic rule is relevant if it is able to be utilised by the semantic query optimizer to increase query efficiency.*

Rule discovery is typically *query driven* or *data driven* (Lowden & Robinson 2002, Shekhar et al. 1993, Siegel et al. 1992, Yu & Sun 1989). In query driven rule discovery, rules are inferred from the restriction clauses of queries arriving at the database and the results they produce. In its simplest form, the method notes when two syntactically different queries produce exactly the same result set (although not necessarily in the same order). The more efficient query will then be substituted whenever the less efficient query arises.

Example 2.3.4. *A simple example of detecting semantic equivalence is the case of three queries, identical except that one is uppercase, another is lower case while the third is a mixture of cases. All three queries produce the same answer. Surprisingly, the detection of such a simple equivalence can be quite important for* *commercial RDBMS. This is because an SQL query identified as being "the same" as one recently parsed, will not be reparsed by the SQL optimizer, but simply submitted directly to the database with the same execution path. Furthermore, if the answer to the previous query is still cached (i.e., in memory), the RDBMS will fetch the answer directly from memory, obviating the need for disk access. If the query is complex, this can result in a significant time saving.*

*The SQL optimizer of at least one major commercial RDBMS does no such textual reformatting. In such a RDBMS, all three queries described above will be judged as being different from one another and each will be separately parsed. Even the addition of extra whitespace will provoke a re-parse (Chan 2006*a*).*

A simple analysis of queries can note which database objects and attributes are actually being queried, creating a focus for the rule discovery exercise. However a subtle problem may arise in that the rules produced may only optimize queries which are the same (or similar to) previous ones received and analyzed. This leaves many potential semantic optimizations unexplored (Shekhar et al. 1993).

In data driven rule discovery, we look primarily at data distribution (Shekhar et al. 1993, Han, Cai & Cercone 1993). Data is analyzed off line in order to discover patterns or correlations that may be formulated into semantic rules. This type of analysis is often described as *data mining* and the application of data mining techniques to discover semantic knowledge for subsequent use in query optimization is described by many researchers in the area (Hsu & Knoblock 2000, Chen 1996, Hsu & Knoblock 1996, Han, Huang, Cercone & Fu 1996, Shekhar et al. 1993, Yu & Sun 1989).

Example 2.3.5. *An analysis of an Employee database reveals that only female employees take maternity leave. The following query is posed:*

```
select *
from   EMPLOYEE
where  GENDER = 'male'
and    MATERNITY_LEAVE = 'yes';
```

If there is no index on columns `MATERNITY_LEAVE` and `GENDER`, this simple query will provoke a full scan of table `EMPLOYEE`, which may be computationally expensive if `EMPLOYEE` is large. Conversely, if we are able to invoke the simple semantic rule that only female employees take maternity leave, this query need never be submitted to the database.

Being independent of any queries, data driven rules may be compiled incrementally without affecting run time performance. The objective of searching for rules in this manner is the assumption that the discovery of, say, a correlation between two column variables, will somehow confer an advantage. So in the context of RDBMS, a reasonable heuristic might be to look for a correlation between an indexed and an unindexed column.

Example 2.3.6. *Consider a database which stores information about ships which dock at a particular port. A table `DOCK` includes columns `SHIP_TYPE` and two columns `ARRIVE` and `DEPART` which record the arrival and departure time respectively of ships that visit. Column `SHIP_TYPE` is indexed while `ARRIVE` and `DEPART` are not indexed. Table `DOCK` is frequently the target of queries which are typically restricted on arrival and departure time. An analysis of data in table `DOCK` reveals that only ships of type `'A'` arrive on Monday or Friday. A semantic rule is formulated:*

$$ARRIVE \in \{\text{'Monday'}, \text{'Friday'}\} \Rightarrow SHIP\_TYPE = \text{'A'}$$

Now queries which ask for arrivals on a Monday or a Friday can add the restriction that the ship must be of type `'A'`. Since column `SHIP_TYPE` is indexed, the addition of the extra predicate increases query efficiency. This is an example of restriction introduction which is described in detail in Section 2.5.3, page 34.

Various researchers identify the problem of finding rules which can actually be utilised by a semantic optimizer (as opposed to simply finding rules, for example, by applying data mining techniques) and advocate the use of heuristics to guide this search (Godfrey et al. 2001, Grant, Gryz, Minker & Raschid 1997, Bell 1996, Siegel et al. 1992, Chakravarthy et al. 1990). Although rule discovery *per se* is beyond the scope of this thesis, we nevertheless list several of the most common heuristics because they foreshadow the type of rule which we employ in our own practical semantic optimizer which we describe in detail in Chapter 4 (page 95). An important feature of the following heuristics is that they depend on both query driven analysis (to identify suitable target tables and columns) and data driven analysis (to formulate the association or correlation into a usable rule).

- If queries against a table T include restrictions on an unindexed column C_i, then look for rules which relate C_i to an indexed column C_j of table T. The objective is to allow an additional constraint based on the indexed column C_j to be introduced. This is an example of *restriction introduction* which is described in detail below in Section 2.5.3, page 34.

- If queries against a table T include restrictions on both columns C_i (unindexed) and columns C_j (indexed) and C_i can be inferred from C_j, then remove C_i from the query. This is an example of *restriction removal* which is described in detail below in Section 2.5.2, page 31.

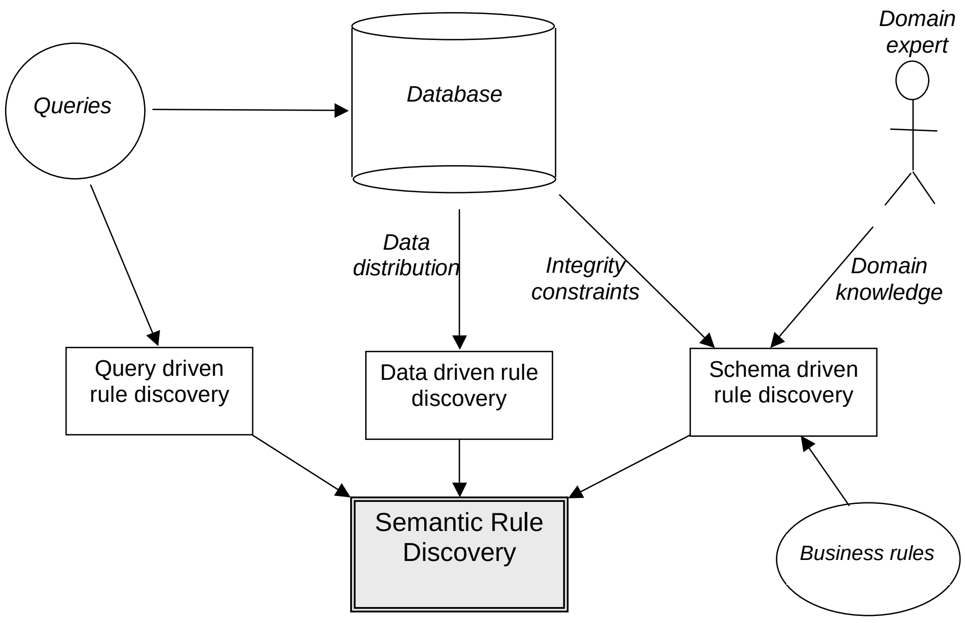


Figure 2.2: **Semantic rule discovery**: Semantic information may be harvested from an analysis of (1) queries (2) data distribution and correlation (3) schema constraints.

- If queries are frequently made against table T which include range restrictions on column C_i, then look for value ranges in column C_i for which there are no satisfying tuples. This is an example of *detection of unsatisfiable queries* which is described in detail below in Section 2.5.1, page 31.

In (Lowden & Robinson 2002), both query driven and data driven techniques are combined. Initially, the join columns cited in equi-join queries become the targets for further data driven analysis. The objective in this case is specifically the discovery of rules to assist with table *joins*, rather than queries on a single attribute. In addition, primary key and corresponding foreign key information is harvested from the DBMS meta-data, again to serve as the starting point for further data analysis. In this case, the objective is the elimination of redundant joins between the table containing the primary key and the table containing the foreign key. This is an example of *join removal* and is described in detail in Section 2.5.4, page 36. We summarize three important approaches to the discovery of semantic information in Figure 2.2 (page 26).

2.3.2 Rule Reliability

In (Godfrey et al. 2001), the authors differentiate between rules which are assumed to be *always true* and have been defined using existing DBMS mechanisms (for example RDBMS *check* constraints) and *soft constraints* which are discovered rules assumed to be true "most" of the time. Soft constraints are themselves divided into *absolute soft constraints* meaning the current state of the database contains no data that violates the rule and *statistical soft constraints* to which a majority of the data comply and a small exception do not. Therefore we have a three level hierarchy defined by a rule's *reliability*; i.e., the probability that it is valid at a particular time:

1. *Schema constraints*: static, true for all data, rule is valid for the lifetime of the schema;
2. *Absolute soft constraints*: dynamic, true for all data, rule is valid for current database state only;
3. *Statistical soft constraints*: dynamic, true for most data, rule is valid for current database state only.

(Hsu & Knoblock 1998) pursue a similar theme of the reliability of a discovered rule. They claim that while most approaches to SQO assume that database semantics are static, in practice they are dynamic. They propose a quantitative metric called *robustness*, which is the probability that a discovered rule is consistent with a database state. A rule has *high robustness* if it is unlikely to become inconsistent after database updates. Robustness of a rule may be estimated from readily available DBMS meta-data. Only rules with high robustness are used for semantic optimization, thereby limiting the cost of re-validating rules.

One intrinsic property of all schema constraints (as we have defined them in Definition 2.3.3, page 20) is that they are created and encoded into the database by human practitioners. It is reasonable then to assume such rules are robust in that they must reflect the intended schema semantics. In contrast, data driven discovered rules are found by the execution of software and we cannot in general assume their robustness. This is in part because we cannot reasonably assume that schema constraints themselves have been enforced for the lifetime of the schema. This apparent anomaly is explained further in Chapter 4 (page 95), but in the meantime we foreshadow the discussion of this problem with an example.

Example 2.3.7. *A large company carries out nightly bulk loads, consolidating sales data into a single data warehouse table. A large summary table,* `SALES`*, includes columns* `PURCHASE_DATE` *and* `SHIP_DATE` *to record the dates when a purchased item is bought and then shipped to a customer. A schema constraint is defined to check that the recorded shipping date is always later than the purchase date:*

```
check SHIP_DATE > PURCHASE_DATE;
```

However, this constraint is relaxed during the bulk load to enhance performance and therefore anomalies where this constraint is violated (for example when

`PURCHASE_DATE` is `null`) are not detected. The `SALES` table is now the target of a data analysis to discover semantic rules which may enhance query performance. Now any rules that utilise the `SHIP_DATE` or `PURCHASE_DATE` columns will be valid with respect to the data, but may be invalid with respect to the intended schema semantics.

When soft constraints are considered for utilisation by a semantic optimizer, some metric is typically used to rank their usefulness. Such rules are often termed *association rules* (Savasere, Omiecinski & Navathe 1995, Park, Chen & Yu 1995, Mannila, Toivonen & Verkamo 1994) and two metrics are typically cited: *support* (a measure of how often the rule occurs in the data set) and *confidence* (a measure of how often the rule is true within the data set) (Agrawal et al. 1993). These metrics are more formally defined as follows. Consider a table T comprising n rows for which an association rule R has been discovered of the form $P_i \Rightarrow Q$ where P_i is some finite conjunction of conditions on the tuples comprising T and Q is the rule's consequent condition. Then:

$$\begin{aligned} support\,(R) &= \frac{|P_i \ or \ Q|}{n} \\ confidence\,(R) &= \frac{|P_i \ and \ Q|}{P_i} \end{aligned}$$

where $|C|$ denotes the number of tuples satisfying condition C. For example, when half the tuples in a target table satisfy either P_i or Q, the potential rule is supported by 50% of the data. A confidence of 90% for the rule indicates the consequent Q can be correctly inferred 90% of the time P_i is true. These metrics are typically chosen prior to the rule discovery phase to limit the complexity of the rule discovery task. So, one would set a minimum support below which the potential rule would not be considered for practical use. Similarly, one would set a minimum confidence (effectively a measure of the strength of the rule) below which the potential rule would not be considered for practical use.

2.3.3 Data Reorganisation

The discovery of semantic information not only leads to query rewrite but can provide compelling reasons to reorganise the storage of data within a database.

Definition 2.3.6. ***Data reorganisation***: *This is the physical relocation of data plus the creation of auxiliary data structures such as clusters, indexes or materialized views, for the purpose of increasing query efficiency.*

One undertakes data reorganisation with the aim of optimizing access to data, primarily data which is stored on disk[5]. The discovery of certain semantic information provides compelling evidence for the creation of (for example) clusters, indexes or materialized views.

Example 2.3.8. *A simple analysis of queries made against a particular database reveals that the most expensive queries are joins between three tables* `A`, `B` *and* `C`*. The database administrator (DBA) decides to co-locate tables* `A`, `B` *and* `C` *and checks that the join columns are indexed. The tables themselves are now subject to further scrutiny with a view to discovering rules to be utilised by a semantic query optimizer.*

Clearly, data reorganisation is a type of query optimization and depends on semantic information for its success. The extraction of this kind of information, along with data reorganisation, is traditionally associated with normal DBA duties.

2.4 Semantic Query Optimization

We now summarise the main components of semantic query optimization and provide a precise definition.

Definition 2.4.1. ***Semantic query optimization****: SQO is the process of uncovering semantic information (from all available sources) plus query rewrite, where the aim is to transform the original query into one which is semantically equivalent but more efficient.*

Most researchers in the field (Aberer & Fischer 1995, Chakravarthy et al. 1990, Godfrey et al. 1996, Gryz et al. 2001, Yu & Sun 1989) use this term to refer to query rewrite. However, we specifically include the activity associated with uncovering semantic information such as rule discovery, in addition to actual query rewrite. Query rewrite is therefore a necessary but not sufficient condition for SQO. Whatever methods are employed to derive semantic information, ultimately this activity results in the actual transformation of the query into a syntactically different but semantically equivalent query.

We summarise the main components of SQO and their interaction in Figure 2.3. Our view is that the harvesting of schema constraints, the discovery of semantic rules via query or data analysis, data reorganisation and query rewrite are all essential aspects of SQO.

[5]This is a well researched topic, beyond the scope of this thesis. We simply note that disk access times are typically orders of magnitude greater than memory access times.

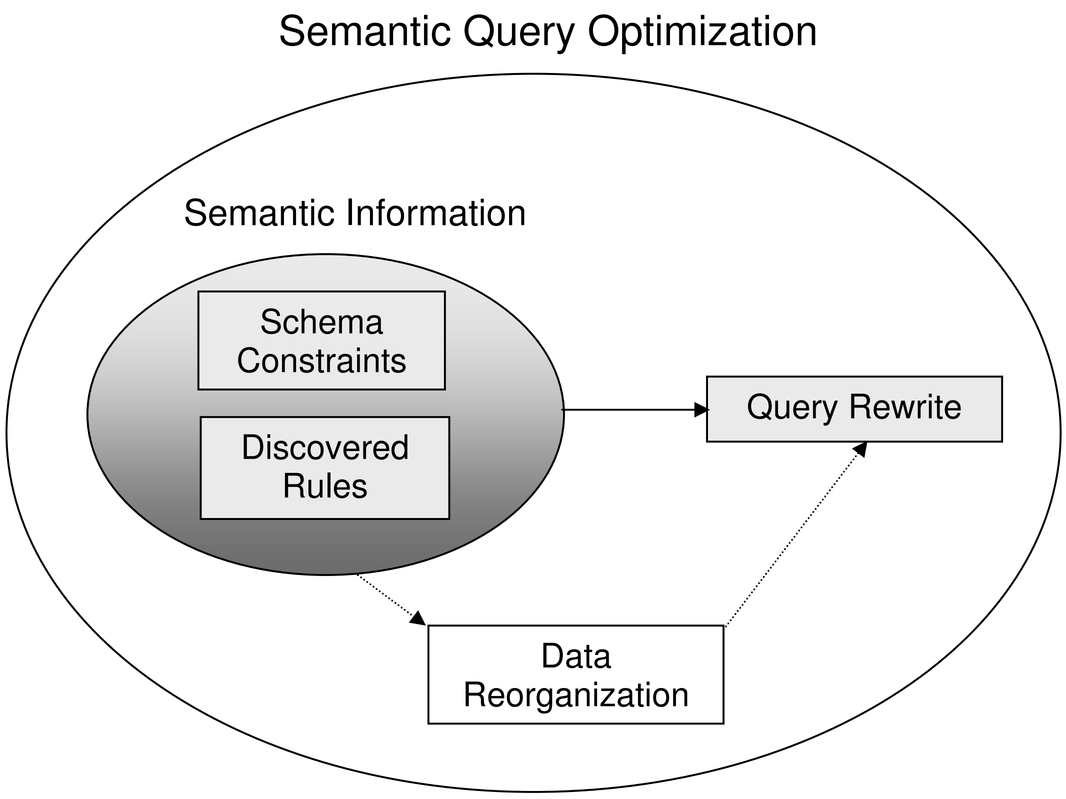


Figure 2.3: **Semantic query optimization**: SQO comprises four major components. (1) Harvesting of schema constraints (2) discovery of semantic rules via query or data analysis (3) data reorganisation (4) query rewrite. The shaded regions comprise a more traditional view of SQO.

2.4.1 Complexity of SQO

When a large rule set exists, which may potentially be used to semantically optimize a query, the problem arises as to which ones are the best to use. The number of transformations suggested by a semantic optimizer can quickly become combinatorially explosive (Siegel et al. 1992). This is known as the *utility problem* (Lowden & Robinson 2002, Han et al. 1993). This is why SQO *per se* is classified as NP-hard (Albrecht et al. 2000, Rishe, Sun & Barton 1995), an unsurprising result given that conventional SQL optimization is also NP-hard (Sun & Yu 1994). Most researchers suggest the use of heuristics to guide the choice of rules and to prune the number of possible transformations to the optimal or near-optimal ones (Shekhar, Srivastava & Dutta 1992, Siegel et al. 1992, Shenoy & Ozsoyoglu 1987, King 1981). Other researchers employ a statistical approach (typically a Chi-square test) to judge the effectiveness of a set of derived rules (Lowden & Robinson 2002, Lowden & Robinson 1999, Sayli & Lowden 1996). In each case the objective is to limit the number of possible transformations available to the semantic optimizer, precluding significant degradation in optimizer efficiency.

2.5 Main Types of SQO

We now summarise and describe the main types of SQO as they have been classified by various researchers (Chomicki 2002, Lowden & Robinson 2004, Lowden & Robinson 2002, Cheng et al. 1999, Lee, Bressan, Goh & Ramakrishnan 1999, Godfrey, Grant, Gryz & Minker 1998, Pang, Lu & Ooi 1991, Chakravarthy et al. 1990, King 1981).

2.5.1 Detection of Unsatisfiable Queries

A query is unsatisfiable if it cannot logically return any rows (Definition 2.2.6, page 18). The detection of such queries is identified as a major advantage by all researchers into SQO (Yoon et al. 1999, Genet & Dobbie 1998, Zhang & Ozsoyoglu 1997, Hsu & Knoblock 1996, Godfrey & Gryz 1996, Illarramendi et al. 1994). For example, (Lowden & Robinson 1999) reports savings made by detecting unsatisfiable queries are an order of magnitude greater than other optimizations. The advantage arises simply because an unsatisfiable query need not be posed to the database at all, resulting in a 100% saving, neglecting the cost of detecting that unsatisfiability.

In the examples that accompany Definition 2.2.6 (page 18), we illustrated query unsatisfiability arising from a logical contradiction within the query text itself and independent of schema semantics (Example 2.2.1, page 17), from a logical contradiction between the query text and the schema semantics (Example 2.2.2, page 18) and because there is no data that satisfies the query restrictions, for the current database state (Example 2.2.3, page 18). In the practical semantic optimizer we describe in Chapter 4 (page 95), all three types of unsatisfiable query are detected.

To our knowledge, no commercial RDBMS implementation performs this optimization. Furthermore, we show in Chapter 5 (page 131) using a cost model, that detecting query unsatisfiability is not costless but may require significant resources at runtime. We confirm this in Chapter 6 (page 153) with the results of our empirical investigation.

2.5.2 Restriction Removal

A query restriction may be deduced to be redundant and its elimination simplifies the query by eliminating the need to process that restriction. We now describe four scenarios where restrictions may be eliminated or at least simplified by consideration of the schema semantics.

- The following example illustrates how restriction removal can arise from a knowledge of the schema meta-data stored as an intrinsic part of the RDBMS.

Example 2.5.1. *Consider a column `COL1` of table `TAB` which is constrained at table creation time to be "`not null`". This prevents any rows being inserted into `TAB` for which `COL1` is null. Therefore whenever the restriction "`COL1 is not null`" appears in queries against `TAB`, it can be eliminated.*

We report empirical results for this type of optimization in Section 6.12 on page 179.

- The following example is similar to the one above and illustrates how a query might be simplified from a knowledge of the schema semantics.

Example 2.5.2. *Consider a column `ID` which is the primary key of table `TAB`. In the following query, the key word "`distinct`" appears in the select clause:*

```
select distinct ID
from   TAB
where  COL1 > 10;
```

The effect of the "`distinct`" key word is to trigger a sort of the result set so that duplicates can be eliminated. However, since column `ID` is the primary key, we can be sure all results returned will be unique. A semantic optimizer might deduce this, saving the cost of the redundant sort.

We report empirical results for this type of optimization in Section 6.10 on page 175.

- The following example illustrates how restriction removal can arise naturally from a knowledge of the range of values actually found in the database for a particular column.

Example 2.5.3. *This example of restriction removal is illustrated in Figure 2.4 (page 33). Consider a table `TAB` with a column attribute `COL1` which appears in the restriction of an SQL query `Q`. A prior knowledge discovery exercise has determined that all values of `COL1` lie on the interval* $[100, 500]$. *Query `Q` is:*

```
select *
from   TAB
where  COL1 >= 0
and    COL1 <= 400;
```

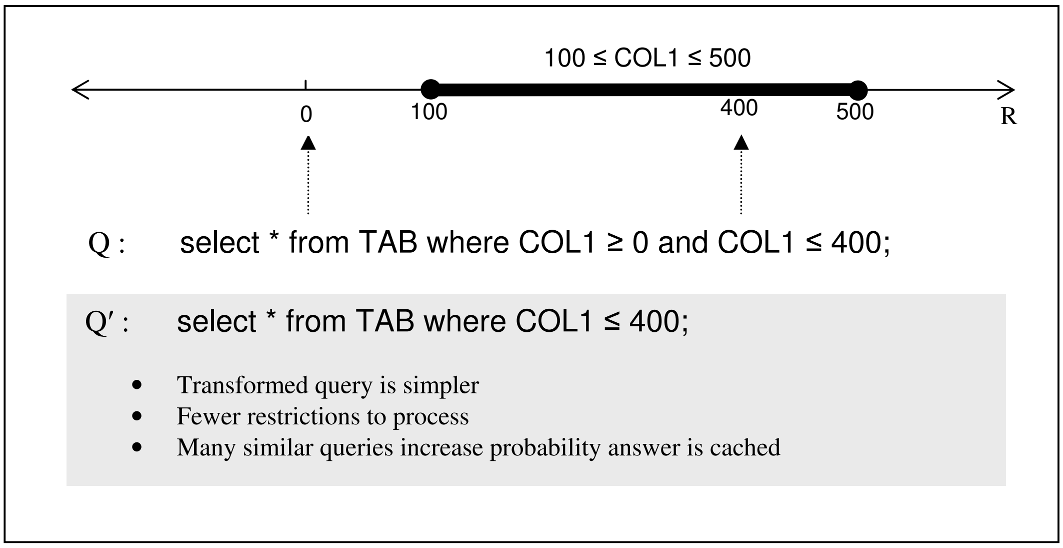

100 ≤ COL1 ≤ 500
0
100
400
500
R
Q : select * from TAB where COL1 ≥ 0 and COL1 ≤ 400;
Q′ : select * from TAB where COL1 ≤ 400;
Transformed query is simpler
Fewer restrictions to process
Many similar queries increase probability answer is cached

Figure 2.4: **Restriction removal**: All values of column `COL1` in table `TAB` lie on the interval $[100, 500]$. A semantic optimizer can apply this knowledge to simplify query `Q` to query `Q′`.

The restriction in `Q` *may be rewritten as the interval* $[0, 400]$. *From a knowledge of these two intervals, a semantic optimizer may deduce that the SQL restriction may be rewritten as "`COL1 <= 400`", recasting the original query into* `Q′`*:*

```
select *
from   TAB
where  COL1 <= 400;
```

The transformed query is simpler in that there are fewer restrictions to process. Furthermore, any similar "out of range" queries on `COL1` *will be recast by a semantic optimizer into the simplified form above. For example, consider queries that restrict* `COL1` *to an interval which subsumes the actual column limits, such as* $[20, 550]$. *All such queries will be recast as the more general query:*

```
select *
from   TAB;
```

This generalisation process increases the chance that the DBMS already has the answer cached[6]*, thereby eliminating the disk activity associated with fetching the answer tuples from disk.*

- The following example illustrates how restriction removal can arise as a result of the application of a rule that has discovered a correlation between an indexed and an unindexed column.

[6]RDBMS typically cache in a memory queue both the parsed SQL query and its result set. Therefore a query which occurs repeatedly will never be aged out the queue.

Example 2.5.4. *Consider a frequently queried table* `TAB`*. A rule discovery exercise has discovered a correlation between unindexed column* `COL1` *and indexed column* `COL2`*. The rule is expressed as:*

```
if COL2 between 'A' and 'E' then COL1 = 'pqr';
```

The following query is posed:

```
select *
from   TAB
where  COL1 = 'pqr'
and    COL2 = 'C';
```

From a knowledge of the second query restriction "`COL2 = 'C'`*" and the above rule, a semantic optimizer may deduce that the first query restriction "*`COL1 = 'pqr'`*" is inferred by the second. Since* `COL2` *is indexed but* `COL1` *is not, the semantic optimizer removes the first restriction and rewrites the query to:*

```
select *
from   TAB
where  COL2 = 'C';
```

We report empirical results for this type of optimization in Section 6.14 on page 182.

2.5.3 Restriction Introduction

A query restriction may imply an additional (redundant) restriction which, when introduced, increases efficiency. This typically occurs when the introduced restriction involves an indexed attribute (Lowden & Robinson 2002, Shenoy & Ozsoyoglu 1987). The following example illustrates how restriction introduction can arise naturally after a rule discovery exercise has discovered a correlation between the columns of a frequently queried table.

Example 2.5.5. *This example of restriction introduction is illustrated in Figure 2.5 (page 35) and is similar to, but the converse of, Example 2.5.4 above. Consider a frequently queried table* `TAB`*. A rule discovery exercise has discovered a correlation between columns* `COL1` *and* `COL2`*. The rule discovery was initiated by three critical observations:*

- `COL2` *has an index of high selectivity;*

Q : select * from TAB where COL1 = ‘xyz’;

<u>Rule</u> : if COL1 in [‘abc’, ‘pqr’, ‘xyz’] then COL2 = ‘A’;

Q′ : select * from TAB where COL1 = ‘xyz’ and COL2 = ‘A’;

Figure 2.5: **Restriction introduction**: The restriction on `COL1` of table `TAB` activates a rule which allows an additional restriction on column `COL2` to be added to the query. Typically, such a rule is applied because an *index* exists on `COL2` but not on `COL1`.

- *`COL1` frequently appears in the restriction clause of SQL queries against table `TAB`;*
- *`COL1` is not indexed.*

The correlation between `COL1` and `COL2` may be expressed as the following rule:

```
if COL1 in ['abc','pqr','xyz'] then COL2 = 'A';
```

Now consider the following query against table `TAB`:

```
select *
from   TAB
where  COL1 = 'xyz';
```

From a knowledge of the query restriction and the above rule, a semantic optimizer may deduce that the additional restriction “`COL2 = 'A'`” may be introduced and recast the query to:

```
select *
from   TAB
where  COL1 = 'xyz'
and    COL2 = 'A';
```

The recast query will invoke the index on `COL2`, returning the answer tuples more quickly.

The rule in Example 2.5.5 above could be expressed as an implication in the form “$\neg X$ *or* Y” and encoded within the RDBMS as a *check constraint.* However this would only achieve the enforcement of the condition at data insert or modification. No commercial RDBMS currently allows the encoding of rules such as these within the DBMS that are invoked at query time[7].

[7]However, *deductive databases* do allow such rules to be defined and invoked in queries in a similar manner to that described above in Example 2.5.5 (Godfrey et al. 1998).

2.5.4 Join Removal

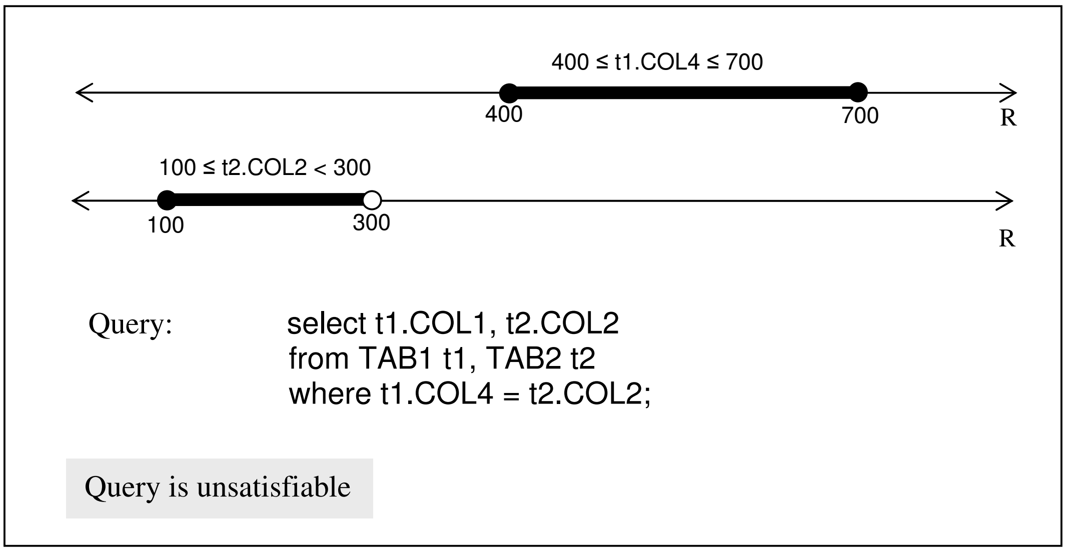
Join removal occurs when a redundant table join is detected and avoided. The join operation in RDB is typically the most expensive (D'Andrea & Janus 1996, Date 2003*b*, Burleson 1994), so it is reasonable to expect its elimination could greatly increase query efficiency. We now describe two scenarios where an unnecessary join operation might be detected by a semantic query optimizer and avoided.

- The following example illustrates how join removal can arise naturally from a knowledge of the range of values actually found in the database for the joined columns.

  Example 2.5.6. *This example of join removal is illustrated in Figure 2.6 (page 36). Consider tables* `TAB1` *and* `TAB2` *with column attributes* `COL4` *and* `COL2` *respectively which frequently appear together as the join columns in equi-join SQL queries. A prior knowledge discovery exercise has determined that all values of* `TAB1.COL4` *lie on the interval* $[100, 300]$ *while all values of* `TAB2.COL2` *lie on the interval* $[400, 700]$. *The following query is posed:*

  ```
  select t1.COL1, t2.COL2
  from   TAB1 t1, TAB2 t2
  where  t1.COL4 = t2.COL2;
  ```

  A semantic optimizer may deduce from a knowledge of the two join column intervals $[100, 300]$ *and* $[400, 700]$ *that this query cannot return any rows; it is unsatisfiable.*



```
select t1.COL1, t2.COL2
from TAB1 t1, TAB2 t2
where t1.COL4 = t2.COL2;
```

Figure 2.6: **Join removal**: The intersection of values for the join columns is null; i.e., they have no values in common. A semantic optimizer may deduce *a priori* the join is unsatisfiable. This query need not be submitted to the database.

- The following example illustrates how join removal can arise from a knowledge of the schema constraints.

  Example 2.5.7. *This example of join removal is illustrated in Figure 2.7 (page 38) which depicts a fragment of a database of sales information. Table* `CUSTOMER` *is a reference table containing customer details and table* `SALES` *records information about products bought by customers. Column* `CUST_ID` *from table* `SALES` *is a non-null foreign key pointing to parent column* `ID` *in table* `CUSTOMER`*, a relationship which is enforced by a constraint stored within the DBMS. The following query* `Q` *is posed:*

  ```
  select c.ID, s.PROD_ID, s.QUANTITY
  from   CUSTOMER c, SALES s
  where  s.CUST_ID = c.ID;
  ```

  By considering the foreign key constraint, a semantic query optimizer may deduce that the equi-join between tables `CUSTOMER` *and* `SALES` *is unnecessary. Exactly the same information is already contained solely within table* `SALES`*, so the join may be eliminated, resulting in the semantically equivalent but simpler query* `Q′`*:*

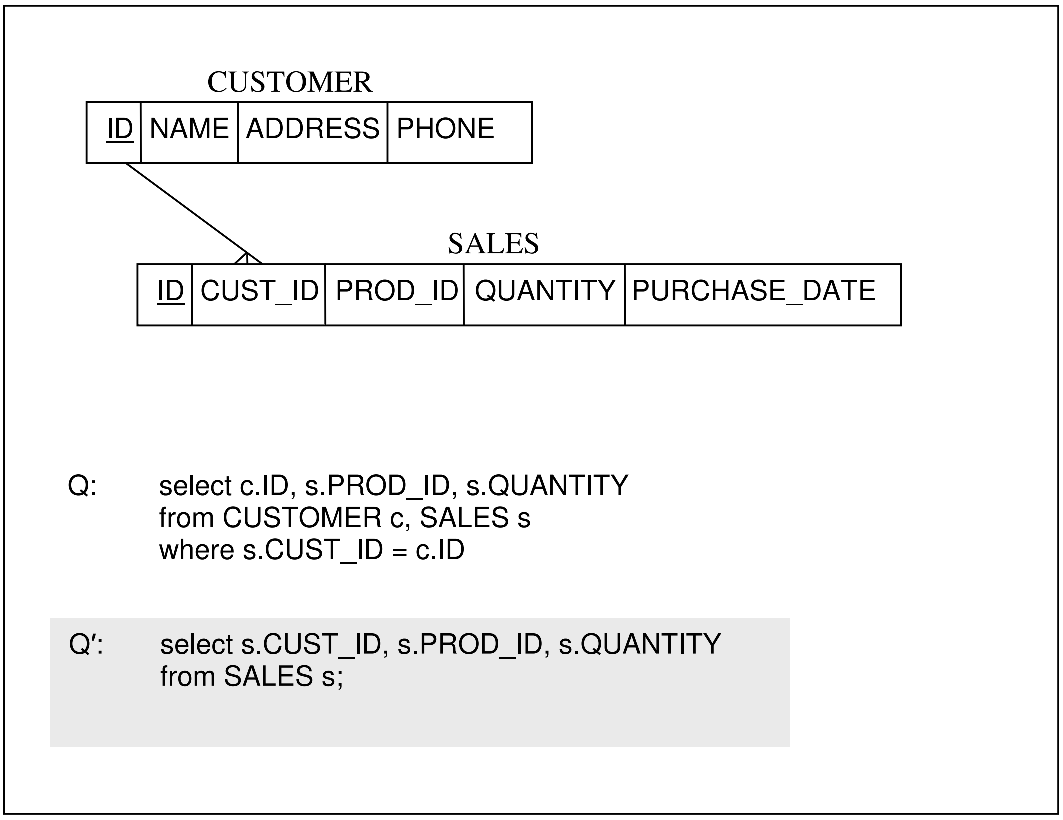
  ```
  select s.CUST_ID, s.PROD_ID, s.QUANTITY
  from   SALES s;
  ```

2.6 SQO in Commercial RDBMS

Our background in the IT industry has led to the observation that SQO is a largely unutilised technique, despite the prevailing view amongst academic researchers that SQO is useful. In (Date 2003*b*), the author comments that semantic optimization could potentially provide much greater performance improvements than more traditional algebraic optimizers, but that few commercial products, if any, do much in the way of semantic optimization. In (Bloesch & Halpin 1997), the authors comment that relational query optimizers ignore many semantic optimization opportunities arising from a knowledge of the schema semantics.

We now look at some of the reasons advanced by other researchers as to why SQO is not routinely employed. This is followed by a brief review of semantic optimization techniques currently employed by a selection of commercial RDBMS.

SQO is known to be useful (Yoon et al. 1999, Date 2003*b*, Hsu & Knoblock 1994). For example (Hsu & Knoblock 1994) reported in 1994 that SQO was achieving average speedups of 20–40% in experiments where semantic knowledge was “hand-coded” into rules able to be utilised by various experimental optimizers.



```
Q:      select c.ID, s.PROD_ID, s.QUANTITY
        from CUSTOMER c, SALES s
        where s.CUST_ID = c.ID

Q′:     select s.CUST_ID, s.PROD_ID, s.QUANTITY
        from SALES s;
```

Figure 2.7: **Join removal**: Column `CUST_ID` is a non-null foreign key pointing to parent column `ID` in table `CUSTOMER`. A semantic query optimizer may deduce that the equi-join between tables `CUSTOMER` and `SALES` is therefore unnecessary. Exactly the same information is already contained solely within table `SALES`, so the join may be eliminated.

Some researchers have claimed significant benefits for empirical studies from their own flavour of SQO (Hsu & Knoblock 1996, Sayli & Lowden 1996) and there are some experimental implementations (Cheng et al. 1999, Godfrey et al. 2001).

In (Cheng et al. 1999), the authors put forward two reasons why SQO has never caught on in the commercial world where most databases are RDBMS:

- SQO is designed for deductive databases where the relatively high cost of applying complex rules (in comparison to much less complex rules in RDB) is more likely to make the extra computational effort of implementing SQO worthwhile;
- CPU speeds are not high enough for the extra computational cost of SQO to be acceptable.

In (Godfrey et al. 2001), the authors consider the role of schema constraints in capturing *business rules* and identify four reasons for SQO techniques not being employed:

- The potential for using schema constraints to capture business rules is only now being realized, so opportunities for SQO have until now seemed limited;

- The expense of checking schema constraints at data insert or update time has limited the use of such constraints, so opportunities for SQO have until now seemed limited;
- Many semantic rules which could potentially be utilised by a semantic query optimizer are simply not discovered;
- Even if a semantic rule is discovered there may be no justification for making it a schema constraint.

The third point reinforces the notion of a *rule discovery phase* (Section 2.3.1, page 21). Without such a phase, only rules that are known *a priori* can be employed. The last point addresses the notion of the *relevance* of the discovered rule (Definition 2.3.5, page 23). A discovered rule may reflect a true correlation between data and is therefore valid, but it may address a part of the domain which is of no interest (for example, because the rule antecedent never appears in a query).

2.6.1 Implemented SQO in Commercial RDBMS

We conclude this section with a brief review of what, if any, semantic optimization techniques are employed in commercial RDBMS[8]. We begin by reiterating our comment from Section 2.5.1 (page 31) that, to our knowledge, no commercial RDBMS implements the detection of unsatisfiable queries.

However, despite a paucity of documentation with respect to SQO in the commercial literature, there are some examples of semantic transformations performed by commercial RDBMS. Some of these are described below and then summarised in Table 2.1 which follows. The following is not a full rigorous review but is intended only to indicate the state of SQO in commercial systems.

- *`DISTINCT` elimination*: Sybase[9] and DB2[10] report the removal of unnecessary `DISTINCT` keywords, potentially saving the cost of a sort operation (Cheng et al. 1999). The following example illustrates this optimization.

  Example 2.6.1. *Consider a table `CUSTOMER` which includes primary key column `ID` and column `NAME`. The following query is posed:*

  ```
  select DISTINCT c.ID, c.NAME
  from   CUSTOMER;
  ```

[8]All of the following information was gathered from publicly available resources, primarily internet websites. The availability of this type of information is subject to the same limitations as other similar information which might be considered commercially sensitive. In each case we provide an internet address as an entry point to the vendor concerned.

[9]Sybase is now marketed under the commercial title of “Adaptive Server Anywhere”. See `http://www.ianywhere.com`.

[10]See `http://www.ibm.com`

Since column `ID` is the primary key of the table, all tuples returned by this query must be distinct. Therefore this query would be recast internally as:

```
select c.ID, c.NAME
from   CUSTOMER;
```

We studied the execution plans produced by the Oracle RDBMS SQL optimizer[11] for cases analogous to the example above and we concluded this semantic optimization of this type is not implemented in the Oracle RDBMS.

- *Optimization of `MIN`, `MAX` functions*: Oracle and Sybase report optimization of the functions `MIN` and `MAX` in the case where an index is present on the target column. The optimization works by scanning the appropriate index in ascending (for the `MIN` function) or descending (for the `MAX` function) order and returning the first row. In this way a redundant sort operation is avoided. Sybase implements this operation transparently while Oracle requires an explicit *hint*[12] to be included with the SQL text to force the use of the appropriate index.

- *Restriction Introduction*: DB2 is reported to implement a type of predicate introduction (Cheng et al. 1999) in that additional joins may be considered in the case of equi-join queries if they can be transitively connected to the join columns of the original query.

  Example 2.6.2. *Consider the following three way join between tables `TAB1`, `TAB2` and `TAB3`:*

```
select t1.COL1, t2.COL1, t3.COL1
from   TAB1 t1, TAB2 t2, TAB3 t3
where  t2.Y = t1.X
and    t3.Z = t2.Y;
```

  A semantic optimizer may deduce that the additional predicate "`t1.X = t3.Z`" can be introduced, producing the following transformed query:

```
select t1.COL1, t2.COL1, t3.COL1
from   TAB1 t1, TAB2 t2, TAB3 t3
where  t2.Y = t1.X
and    t3.Z = t2.Y
and    t1.X = t3.Z;
```

[11] See (Chan 2006*d*) for a concise explanation of manifesting the execution path chosen by the Oracle SQL optimizer.

[12] In the Oracle RDBMS, hints may be included within SQL query text to force the use of particular access paths. Further discussion of hints is beyond the scope of this thesis. See for example (Fogel & Lane 2006*b*).

This might result in increased efficiency if the new join can be used early in the execution plan to reduce the cardinality of the result set.

- *Redundant Join Removal*: Sybase and DB2 report the removal of redundant joins in certain circumstances, although only Sybase documents this explicitly with an example similar to Example 2.5.7 above (page 37). We studied the execution plans produced by the Oracle RDBMS SQL optimizer to look for evidence of redundant join removal and concluded this semantic optimization is not implemented in Oracle.

- *Other optimizations*: Sybase report several other optimizations such as the elimination of the redundant clause "`or 1 = 0`". We investigated this specific example with Oracle and despite reporting an execution plan which suggested this redundancy had been eliminated, actual measurement of execution times for medium to large tables strongly suggested this redundant clause provoked a full table scan. Sybase report the rewrite of single "`IN`" clauses such as "`COL1 in (100)`" which is recast as "`COL1 = 100`". Our experiments with Oracle show this simple rewrite is in fact implemented but is undocumented.

Type of SQO	**RDBMS**	**Comment**
Restriction removal	DB2, Sybase	"`distinct`" elimination.
Restriction removal	Oracle, Sybase	Optimization of `MIN`, `MAX` function.
Restriction removal	Sybase	Remove clause "`or 1 = 0`"
Restriction introduction	DB2	Additional join clauses added.
Join removal	DB2, Sybase	See Example 2.5.7, page 37.
Other	Oracle, Sybase	Rewrite "`COL1 in (100)`" to "`COL1 = 100`"

Table 2.1: **Implemented SQO in commercial RDBMS**: This table summarises SQO techniques currently employed by three commercial RDBMS: *Oracle*, *DB2* and *Sybase*.

2.7 Summary

In this chapter we described related research in the field of SQO. The main contributions of this chapter include the following:

- We present a concise set of definitions of important terms used by other researchers in the field of SQO, including a precise and expanded definition of the principle term "semantic query optimization" (Sections 2.2 and 2.4).

- We describe the main sources of semantic information including *schema constraints* which are already part of the meta-data stored and maintained by the RDBMS itself and *discovered rules* which are typically formulated via an analysis of queries (query driven rule discovery) or data (data driven rule discovery) (Section 2.3).

- We show how semantic rules may be differentiated according to their *reliability*. Rules which are *static* such as schema constraints may be applied for the lifetime of the schema, while *dynamic* rules such as those formulated by the discovery of correlations in data may need to be revalidated or marked as invalid whenever data updates occur. This may have a significant impact on database performance (Section 2.3).

- We set out the main types of SQO as they have been classified by other researchers and provide concrete examples of how such optimization opportunities can arise in practice (Section 2.5).

- We consider SQO in the context of commercial RDBMS and briefly review a small set of semantic optimizations which are implemented in some commercial RDBMS (Section 2.6).

Chapter 3

An Algebra of Intervals

3.1 Introduction

In Section 1.3.2 of the previous chapter, we explained that while SQL optimizers apply the rules of relational algebra, they lack a *semantic* reasoning engine (Hsu & Knoblock 2000, Godfrey et al. 1996) and, apart from the very limited cases described in Section 2.6.1, are therefore unable to use semantic information to influence the choice of a suitable execution path. The main objective of this current chapter is to build the theoretical foundation for our reasoning engine, which we describe in detail in the following chapter. We now describe an *interval* data type around which we build an *interval algebra* which we show is analogous to Boolean Algebra and which forms the foundation of our reasoning engine. We accept the Boolean Algebra as axiomatic (Pohl & Shaw 1986) along with certain axioms concerned with the *total ordering* of data types (Gemignani 1990). We use the interval data type to succinctly capture the notion of *the valid or legal range of values a variable may assume.* Our interval is a generalisation of the notion of an interval on the Real number line (Muñoz & Lester 2005). We define three basic operations: *conjunction*, *disjunction* and *negation* for the interval data type and show how the resulting algebra can be used to reason about values enclosed by the interval. We provide an appealing graphical interpretation which depicts the interaction of intervals. We focus on intervals built from the three atomic data types found in relational databases: *numeric*, *string* and *date.*

We then define a further data type, the *interval list*, which is *a collection of disjoint intervals.* We extend our definitions of conjunction, disjunction and negation to the interval list structure and show how these operations are *closed* with respect to the interval list.

The remainder of this chapter is organised as follows.

- We begin by setting out our basic assumptions with regard to the *ordering of instances* of the data types we deal with and introduce constants to represent *minus infinity* and *plus infinity* (Section 3.2).
- We then define two data types: *limits* (Section 3.3) and *bounds* (Section 3.4) that comprise our interval data type.
- We define precisely the notion of an *interval* and introduce the special intervals the *infinite interval* and the *null interval* (Section 3.5).
- With respect to the interval data type, we define the binary operations *conjunction* (Section 3.6) and *disjunction* (Section 3.7).
- We then define a new data type, the *interval list*, which is a collection or list of disjoint intervals (Section 3.8).

- We describe how to negate an interval (Section 3.9).
- We show how the infinite interval and the null interval act as *identity elements* for the conjunction and disjunction operations (Section 3.10).
- We define two further binary operations for intervals, *subsumption* and *implication* (Section 3.11).
- We then extend to interval lists the disjunction operation (Section 3.12), the conjunction operation (Section 3.13) and the negation operation (Section 3.14).
- We define the special interval lists the *infinite interval list* and the *null interval list* (Section 3.15).
- We then extend the operations *subsumption* and *implication* to interval lists (Section 3.16).
- We conclude by listing the main contributions of the Chapter (Section 3.17).

3.2 Basic Assumptions and Working Definitions

In this section we set out our basic assumptions and provide working definitions for some important terms which we employ throughout this chapter. We first describe how modern programming languages *overload* the Boolean comparison operators. Then we introduce a generic data type and state some important assumptions about how this data type can be deterministically ordered. Finally we introduce constants for *plus infinity* and *minus infinity* and show how these are fully usable in the context of a programming language, provided they have a restricted and well defined meaning.

3.2.1 Overloading of Boolean Comparison Operators

Throughout this chapter we shall often use intervals on the Real number line for examples. But we might just as well use the *string* or *date* data types. The key property of these data types which we utilise is that each has a well defined deterministic ordering. This is usually accepted without question and is universally implemented by programming languages and by relational database management systems (RDBMS) in particular.

In fact, the ordering method is different for each different data type. When numbers are compared, numeric ordering is used. When strings are compared, lexigraphic ordering is used. When dates are compared, date ordering is used.

For this reason, in relational databases and most programming languages, the Boolean comparison operators “$<, >, \leq, \geq, =$” are said to be *overloaded* because their validity depends on the correct ordering method being invoked according to data type. We will also overload these symbols as we proceed through the chapter and define unambiguous orderings for a number of different data types. In each case, the type of ordering we mean will be evident from the context.

3.2.2 A Generic Data Type

When we do not need to distinguish between data types that may be *totally ordered* (Gemignani 1990) (for example, the three atomic data types *numeric*, *string* and *date*) we will employ the generic notation T and refer to instances of T using $t \in T$. When we employ this notation, we mean any data type that is able to be totally ordered can be substituted for T without changing the validity of the statement.

3.2.3 Ordering of Instances $t \in T$

We take as axiomatic the existence of a well defined total ordering (Gemignani 1990) for the data types to which our generic data type T refers. Therefore any set of instances of T can be deterministically ordered from left to right and we can meaningfully employ the Boolean comparison operators “$<, >, \leq, \geq, =$” in the conventional way. Consider a set of n instances of T which we have ordered, with the leftmost instance having the lowest rank and the rightmost instance the highest rank. We can make use of the Boolean comparison operator “$<$” to express this ordering:

$$t_1 < t_2 < \cdots < t_n$$

3.2.4 Comparing Instances $t \in T$

We now define a deterministic function “$compare_T$” which we will use to compare the rank of any two instances $t_1, t_2 \in T$. We design our function to return -1 if the rank of t_1 is less than t_2, 0 if they are the same rank and 1 if t_1 has a higher rank than t_2.

Algorithm 3.2.1. ***Comparing Instances*** $\mathbf{t_1, t_2 \in T}$*: We define using pseudo-code the following deterministic function* $compare_T$*:*

```
$compare_T\ (t_1, t_2)$ $return\ integer\ is$
    $ret \leftarrow 1;$
    $if\ \ t_1 = t_2\ \ then\ \ ret \leftarrow 0$
```

```
        elsif $t_1 < t_2$ then ret $\leftarrow$ –1
        endif;
        return ret;
```

The Boolean comparison operators “=” and “<” in the above definition are overloaded as we explain in Section 3.2.1 (page 45). Although we are simply comparing ranks, the method used to decide their rank depends on the data type to which we are referring.

From now on, whenever we use the Boolean comparison operators with a particular data type, we will be doing so only because we have already defined a deterministic function “*compare*” for that particular data type which returns appropriately the values −1, 0 or 1, in the manner set out above in Algorithm 3.2.1.

3.2.5 Representing Minus and Plus Infinity

We wish to incorporate the notion of *minus infinity* and *plus infinity* into our function $compare_T$ defined above in Algorithm 3.2.1. When we use these terms we intend only to convey the idea of a value which would always be the first(last) value were it added to any ordered list of values $t \in T$. We will use the special symbol “$MINF$” to denote minus infinity and the symbol “$PINF$” to denote plus infinity. We treat these two special values simply as constants and add them to whatever set of values comprise the domain we are considering.

Definition 3.2.1. ***MINF****: This is the lowest ranked value in any set of values containing at least one occurrence of constant MINF. That is:*

$$MINF \le t \qquad \forall t \in T$$

Definition 3.2.2. ***PINF****: This is the highest ranked value in any set of values containing at least one occurrence of constant PINF. That is:*

$$PINF \ge t \qquad \forall t \in T$$

Using the above definitions, we adjust our algorithm for $compare_T$ to allow for the possibility of these constants occurring.

Algorithm 3.2.2. ***Comparing Instances*** $t_1, t_2 \in T$ ***when*** $MINF, PINF \in T$*: We define using pseudo-code the following deterministic function* $compare_T$*. The following function is optimal in the sense that it requires the minimum number of comparisons to achieve its objective:*

```
    $compare_T$ $(t_1, t_2)$ return integer is
        ret $\leftarrow$ 1;
```

```
$if\ \ t_1 = t_2\ then\ \ ret \leftarrow 0$
$elsif\ \ t_1 = MINF\ \ then\ \ ret \leftarrow -1$
$elsif\ \ t_1 = PINF\ \ then\ \ ret \leftarrow 1$
$elsif\ \ t_2 = PINF\ \ then\ \ ret \leftarrow -1$
$elsif\ \ t_2 = MINF\ \ then\ \ ret \leftarrow 1$
$elsif\ \ t_1 < t_2\ \ then\ \ ret \leftarrow -1$
$end\ if;$
$return\ \ ret;$
```

3.3 Limit Operators

We now employ the familiar parenthesis and bracket notation to help define *exclusive* and *inclusive* (respectively) upper and lower bounds for an interval. It will be useful for the definitions that follow to consider these symbols as *operators* that operate on instances $t \in T$.

Definition 3.3.1. ***Limit operator**: We define four limit operators as set out below in Table 3.1.*

Limit Operator	Description
[	Left inclusive limit
(	Left exclusive limit
]	Right inclusive limit
)	Right exclusive limit

Table 3.1: **The four limit operators**.

We employ these symbols as operators which may only operate on instances $t \in T$ where we have a well defined ordering as described above in Algorithm 3.2.2. We have exactly four limit operators: two *left limit operators* "[" and "(" plus two *right limit operators* "]" and ")".

3.3.1 Comparing Limits

We now show how the limit operators can be deterministically ordered. Our ordering is intuitive and is a generalisation of ordering inclusive and exclusive limits on the Real number line. In fact, it is not really the limit operators themselves that are being ordered but the result of their operating on a data type instance. Nevertheless, it turns out to be convenient to treat the operators as if they enjoy an independent existence.

We wish to compare two limit operators, say, l_1 and l_2. Since there are exactly four limit operators, we have exactly 16 cases to consider. Table 3.2 lists the 16 cases and the result of applying the $compare_{limit}$ function.

Limit l_1	Limit l_2	$compare_{limit}(l_1, l_2)$
(	(	0
(	[	1
(	)	1
(	]	1
[	(	–1
[	[	0
[	)	1
[	]	0
)	(	–1
)	[	–1
)	)	0
)	]	–1
]	(	–1
]	[	0
]	)	1
]	]	0

Table 3.2: **Rank of limit operators**: Comparing the rank of just the limit operators alone gives rise to 16 cases. Note that there are six cases (rather than four) where the function returns 0; i.e., where l_1 and l_2 are considered to have the same rank. The ranking we apply is intuitive but rigorous and is a generalisation of ordering inclusive and exclusive limits on the Real number line.

Algorithm 3.3.1. ***Comparing Limit Operators**: Using Table 3.2, we now define using pseudo-code a deterministic function "$compare_{limit}$" which we will use to compare the rank of any two limits l_1 and l_2. The following is an optimal implementation of Table 3.2 in the sense that it uses a minimum number of comparisons.*

```
$compare_{limit}(l_1, l_2)$ return integer is
    $ret \leftarrow 1$;
    if $l_1 = l_2$ then $ret \leftarrow 0$
    elsif $l_1 = ($ then $ret \leftarrow 1$
    elsif $l_1 = )$ then $ret \leftarrow -1$
    elsif $l_2 = ($ then $ret \leftarrow -1$
    elsif $l_2 = )$ then $ret \leftarrow 1$
    else $ret \leftarrow 0$
    end if;
    return $ret$;
```

Algorithm 3.3.1 defines an ordering method which is neither numeric nor lexigraphic nor date. The 16 cases set out above in Table 3.2 are not at all arbitrary but are dictated by considering what makes sense for a data type with a deterministic total ordering such as the Real numbers. Note that there are six cases (rather than four) where the $compare_{limit}$ function returns 0; i.e., where l_1 and l_2 are considered to have the same rank. The extra two cases arise because we define "[" to have the same rank as "]". This captures the intuitive notion that "[t" and "t]" bracket the same instance $t \in T$.

3.3.2 Negating Limits

Although we will not utilise this result until Section 3.4.6 below (page 54), we now define the *negation* of a limit. We will write the negation of a limit operator l as l'.

Definition 3.3.2. ***Negation of Limit***: *There are just four cases to consider and these are set out in Table 3.3.*

Limit: l	Negation of Limit: l'
[	)
(	]
]	(
)	[

Table 3.3: **The four limit operators and their negation**.

Algorithm 3.3.2. ***Negating Limit Operators***: *Using Table 3.3, we now define using pseudo-code an algorithm "neg_{limit}" to return the negation of a limit l.*

```
$neg_{limit}\,(l)$ return limit is
    ret $\leftarrow$;
    if  $l$ = [  then  ret $\leftarrow$ )
    elsif  $l$ = (  then  ret $\leftarrow$ ]
    elsif  $l$ = ]  then  ret $\leftarrow$ (
    elsif  $l$ = )  then  ret $\leftarrow$ [
    end if;
    return  ret;
```

3.3.3 Notation for Limits

We will sometimes require meta-symbols to denote either of the two left limits, the two right limits, or their negation. Table 3.4 sets out these symbols:

Meta-symbol	Denotation	Description
$\langle$	[or (	left limit
$\rangle$	] or)	right limit
$\langle'$	] or)	negated left limit
$\rangle'$	[or (	negated right limit

Table 3.4: **Meta-symbols for the four limit operators and their negation**.

3.4 Bounds

The result of operating on an instance $t \in T$ with any limit operator is a *bound*. There are exactly four types of bound: two *left bounds* and two *right bounds* [1].

Definition 3.4.1. ***Bound**: Consider two instances $t_1, t_2 \in T$ and a variable $x \in T$. Using the four limit operators defined in Section 3.3 (page 48), we may form four bounds which have the interpretation described in Table 3.5.*

Limit	Value	Bound	Description	Interpretation
[	t_1	$[t_1$	left inclusive bound	$\{x: \ x \geq t_1\}$
(	t_1	$(t_1$	left exclusive bound	$\{x: \ x > t_1\}$
]	t_2	$t_2]$	right inclusive bound	$\{x: \ x \leq t_2\}$
)	t_2	$t_2)$	right exclusive bound	$\{x: \ x < t_2\}$

Table 3.5: **Four bounds and their interpretatio**n.

Example 3.4.1. *Consider Figure 3.1 which depicts four bounds where the data type is the Real numbers $\mathbb{R}$. The bounds consist of instance $r \in \mathbb{R}$ plus an associated limit operator. We use the familiar notation of open and closed circles to visualize exclusive and inclusive bounds and depict the meaning of these bounds by plotting them on the Real number line.*

3.4.1 Notation for Bounds

We see from Definition 3.4.1 above that each bound consists of a *limit operator* and a *value*. The pair "`limit,value`" form a tuple. However we will not introduce a meta-symbol to bracket these pairs or a meta-symbol to delineate them. We will instead simply write them alongside each other.

For example, consider an arbitrary bound *B*. We can decompose this into a limit operator l and a value v. We may write:

$$B = lv \qquad \text{where} \quad l \in \{\,(,\,),\,[,\,]\,\}, \quad v \in T$$

[1]It might be more orthodox to refer to these as *upper bounds* and *lower bounds* respectively. However we use the descriptors *left* and *right* to avoid ambiguity in the descriptions of the algorithms for conjunction and disjunction which follow.

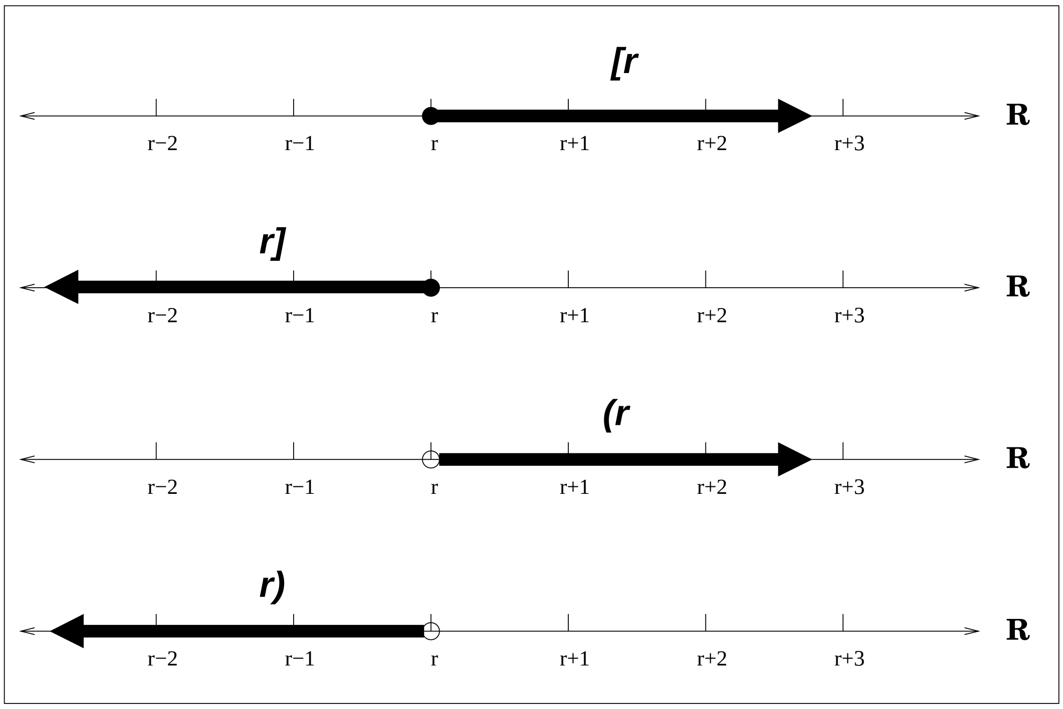


Figure 3.1: **Bounds on the Real number line**: r is an arbitrary point on the Real number line. The solid arrows depict the interpretation of the bounds formed when the four limit operators are applied in turn to point r. There are two left bounds: $(r$, $[r$ and two right bounds: $r)$, $r]$. We use the familiar notation of open and closed circles to visualize exclusive and inclusive bounds respectively.

We place no significance on the order we write the limit and value. When we write bounds with actual atomic instances (for example in Table 3.5) we simply follow the convention of writing left bounds in the order "`limit value`" and right bounds in the order "`value limit`".

3.4.2 A Bound is a Logical Assertion

Consider the interpretation column of Table 3.5 (page 51). This highlights the fact that a bound is a contracted form of logical assertion where the variable is implicit, utilising the Boolean operators "$>$" and "$\geq$" for the left bounds while "$<$" and "$\leq$" are utilised for the right bounds.

3.4.3 Comparing Bounds

We now show, given our target data type T (which has a well defined total ordering) and ordering of limit operators described in Algorithm 3.3.1 (page 49), how we may unambiguously compare the rank of any two bounds. In the algorithms that follow, we consider two arbitrary bounds B_1 and B_2 which consist, respectively, of instances $t_1, t_2 \in T$ plus their associated limit operators l_1 and l_2. That is:

$$B_1 = l_1 t_1$$
$$B_2 = l_2 t_2$$

Algorithm 3.4.1. ***Comparing Bounds**: We define a deterministic function $compare_{bound}$ which compares the rank of any two bounds B_1 and B_2 and is given by the following pseudo-code:*

```
$compare_{bound}\,(B_1, B_2)$ return integer is
    $ret \leftarrow 0;$
    if $t_1 < t_2$ then $ret \leftarrow -1$
    elsif $t_1 > t_2$ then $ret \leftarrow 1$
    else $ret \leftarrow compare_{limit}\,(l_1, l_2)$
    end if;
    return ret;
```

Our comparing of bounds proceeds by first comparing the data type instances t_1 and t_2. Implicitly, we call the $compare_T$ function associated with data type T (Algorithm 3.2.1, page 46), which by our definition is guaranteed to exist. Only if the two data type instances have equal rank do we call $compare_{limit}$ (Algorithm 3.3.1, page 49). We note that $compare_{limit}$ is quite independent of data type T whereas $compare_T$ is entirely dependent on its method implementation.

3.4.4 Infinite Bounds

In Section 3.2.5 (page 47) we defined constants "$MINF$" and "$PINF$" which would always be the first or last values respectively were they included in any ordered list of values. We now use these constants along with the inclusive limit operators "[" and "]" to define two infinite bounds: the *infinite left bound* and the *infinite right bound.* When we use these terms, we mean simply that the infinite left bound would always be the leftmost bound were it to be included in any ordered list of bounds. Similarly, the infinite right bound would always be the rightmost bound were it to be included in any ordered list of bounds. From now on we will use the symbols "MIB" and "PIB" to denote the infinite left bound and infinite right bound respectively.

Definition 3.4.2. ***Infinite left bound**: This is the bound formed by applying the left inclusive limit operator [to the constant $MINF$. That is:*

$$MIB \equiv [MINF$$

Definition 3.4.3. ***Infinite right bound**: This is the bound formed by applying the right inclusive limit operator* $]$ *to the constant* $PINF$. *That is:*

$$PIB \equiv PINF]$$

3.4.5 Functions *lower* and *higher*:

We now define two useful auxiliary functions for bounds which we will utilise later in this chapter. We use Algorithm 3.4.1 (page 53) to define a function "*lower*" to return the lower of two bounds and a function "*higher*" to return the higher of two bounds. In the following algorithms, the Boolean operators ">" and "<" are overloaded. The comparison which is performed implicitly utilises Algorithm 3.4.1 (page 53).

Algorithm 3.4.2. ***Lower Bound**: We define the following pseudo-code function to return the lower of two bounds* B_1 *and* B_2*:*

```
$lower\,(B_1, B_2)$ $return\ bound\ is$
    $ret \leftarrow B_1;$
    $if\ \ B_1 > B_2\ \ then$
      $ret \leftarrow B_2$
    $end\ if;$
    $return\ \ ret;$
```

Algorithm 3.4.3. ***Higher Bound**: We define the following pseudo-code function to return the higher of two bounds* B_1 *and* B_2*:*

```
$higher_{bound}\,(B_1, B_2)$ $return\ bound\ is$
    $ret \leftarrow B_1;$
    $if\ \ B_1 < B_2\ \ then$
      $ret \leftarrow B_2$
    $end\ if;$
    $return\ \ ret;$
```

3.4.6 Negating Bounds

We now consider what happens when we negate a left or a right bound. Consider an arbitrary left bound B_L consisting of a left limit $[$ and value v_L. We introduce a variable $x \in T$, write this bound as an inequality, then negate it:

$$\begin{aligned} B_L &= [v_L \\ &= \{x : \ x \geq v_L\} \\ \neg B_L &= \neg\{x : \ x \geq v_L\} \\ &= \{x : \ x < v_L\} \\ &= v_L) \end{aligned}$$

So the left bound has become a right bound but the value v_L remains the same. The new bound is formed because the limit operator has been negated, according to Definition 3.3.2 (page 50).

Similarly, beginning with right bound $B_R = v_R]$ and negating it:

$$\begin{aligned} B_R &= v_R] \\ &= \{x : \ x \leq v_R\} \\ \neg B_R &= \neg\{x : \ x \leq v_R\} \\ &= \{x : \ x > v_R\} \\ &= (v_R \end{aligned}$$

So the right bound has become a left bound but the value v_R remains the same. The new bound is formed because the limit operator has been negated, according to Definition 3.3.2 (page 50).

These observations lead to the following definition.

Definition 3.4.4. ***Negation of Bound**: Consider an arbitrary bound B consisting of limit l and value v. That is, $B = lv$. We write the negation of limit l as l'. The negation of B is written $\neg B$ and is defined by:*

$$\neg B = l'v$$

3.4.7 Conjunction and Disjunction of Bounds

We now show that the Boolean conjunction of two *left* bounds is equivalent to the *higher* of those bounds. Similarly, the Boolean conjunction of two *right* bounds is the *lower* of those bounds.

For Boolean disjunction of bounds, the converse is true. The Boolean disjunction of two *left* bounds is equivalent to the *lower* of those bounds. Similarly, the Boolean disjunction of two *right* bounds is the *higher* of those bounds.

We first consider the Boolean conjunction of two *left* bounds and prove the correct result is the higher of these bounds. We then state the other theorems by analogy.

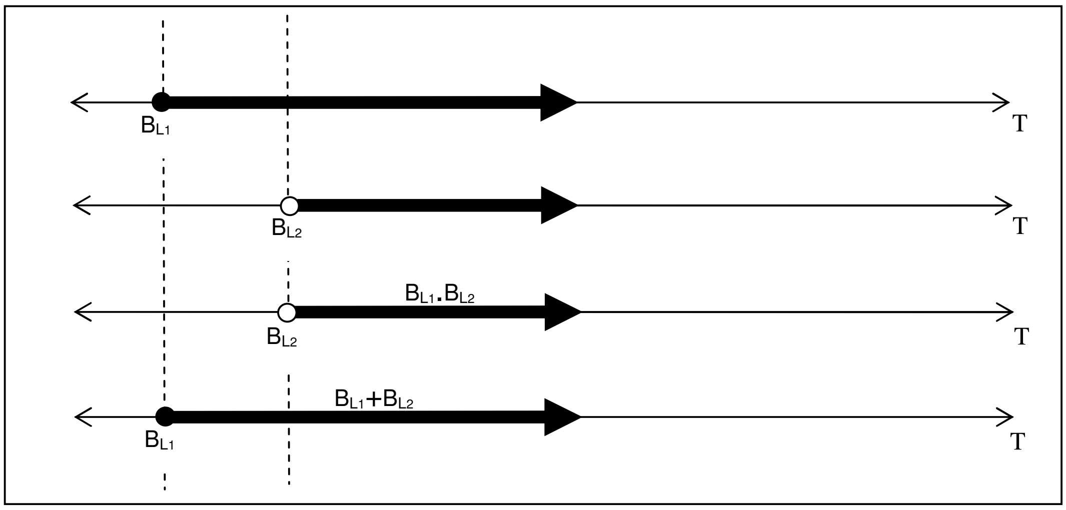


Figure 3.2: **Conjunction and disjunction of left bounds**: The Boolean conjunction of two left bounds $B_{L_1} \cdot B_{L_2}$ is logically equivalent to the higher of the two bounds $higher_{bound}(B_{L_1}, B_{L_2})$. The Boolean disjunction of two left bounds is logically equivalent to the lower of the two bounds.

We complete this section by showing that the disjunction of a left and right bound always yields "`true`" whenever the left bound is less than or equal to the right bound.

Theorem 3.4.1. ***Conjunction of Left Bounds**: This is illustrated in Figure 3.2. Consider two arbitrary left bounds B_{L_1} and B_{L_2} over domain T. Then the Boolean conjunction of the bounds $B_{L_1} \cdot B_{L_2}$ is given by:*

$$B_{L_1} \cdot B_{L_2} = higher_{bound}(B_{L_1}, B_{L_2})$$

***Proof**: Inspection of Algorithm 3.4.1 (page 53) shows the only time limits are compared is when the values are equal to a value v_L say. So it is sufficient to check $B_{L_1} \cdot B_{L_2}$ is given by the higher of the two bounds when this condition occurs. There are exactly four possibilities and these are set out in Table 3.6. In each case, the equivalent logical assertion is correctly given by the higher left bound, as dictated by Algorithm 3.4.3.*

$B_{L_1} \cdot B_{L_2}$	Assertion	$higher_{bound}$
$[v_L \cdot [v_L$	$(x \geq v_L) \cdot (x \geq v_L) = x \geq v_L$	$[v_L$
$[v_L \cdot (v_L$	$(x \geq v_L) \cdot (x > v_L) = x > v_L$	$(v_L$
$(v_L \cdot [v_L$	$(x > v_L) \cdot (x \geq v_L) = x > v_L$	$(v_L$
$(v_L \cdot (v_L$	$(x > v_L) \cdot (x > v_L) = x > v_L$	$(v_L$

Table 3.6: **Conjunction of left bounds when the values are equal**: This depends only on the comparison of the left limit operators. In each case, the equivalent logical assertion is correctly given by the higher left bound, as dictated by Algorithm 3.4.3.

Theorem 3.4.2. ***Conjunction of Right Bounds**: This is illustrated in Figure 3.3. Consider two arbitrary right bounds B_{R_1} and B_{R_2} over our domain T. Then the*

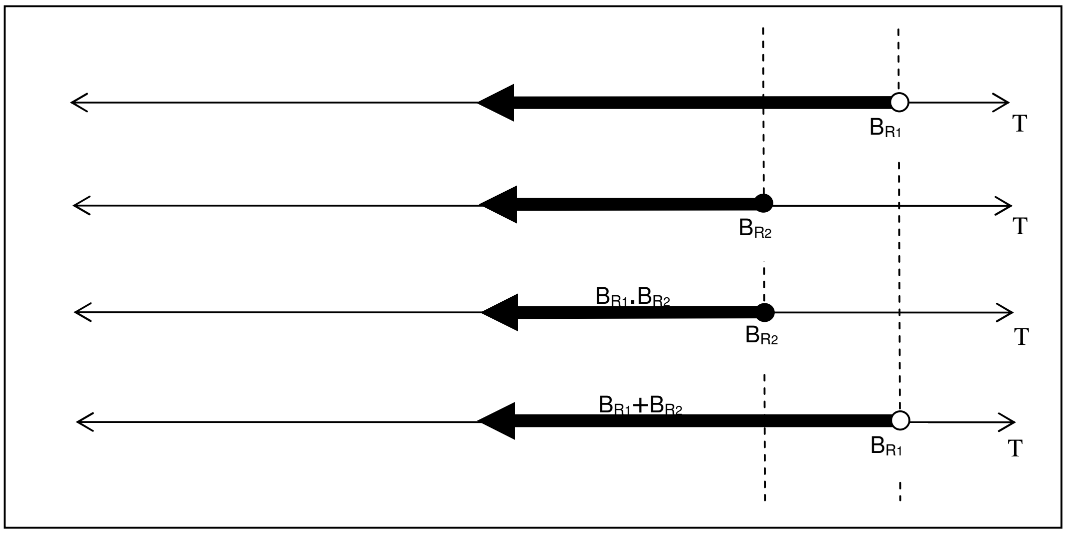


Figure 3.3: **Conjunction and disjunction of right bounds**: The Boolean conjunction of two right bounds $B_{R_1} \cdot B_{R_2}$ is logically equivalent to the lower of the two bounds $lower(B_{R_1}, B_{R_2})$. The Boolean disjunction of two right bounds is logically equivalent to the higher of the two bounds.

Boolean conjunction of the bounds $B_{R_1} \cdot B_{R_2}$ is given by:

$$B_{R_1} \cdot B_{R_2} = lower(B_{R_1}, B_{R_2})$$

Proof: *This proof is analogous to the proof of Theorem 3.4.1 (page 56) above.*

Theorem 3.4.3. ***Disjunction of Left Bounds***: *This is illustrated in Figure 3.2. Consider two arbitrary left bounds B_{L_1} and B_{L_2} over our domain T. Then the Boolean disjunction of the bounds $B_{L_1} + B_{L_2}$ is given by:*

$$B_{L_1} + B_{L_2} = lower(B_{L_1}, B_{L_2})$$

Proof: *This proof is analogous to the proof of Theorem 3.4.1 (page 56) above.*

Theorem 3.4.4. ***Disjunction of Right Bounds***: *This is illustrated in Figure 3.3. Consider two arbitrary right bounds B_{R_1} and B_{R_2} over our domain T. Then the Boolean disjunction of the bounds $B_{R_1} + B_{R_2}$ is given by:*

$$B_{R_1} + B_{R_2} = higher_{bound}(B_{R_1}, B_{R_2})$$

Proof: *This proof is analogous to the proof of Theorem 3.4.1 (page 56) above.*

We complete this section by showing that the disjunction of a left and right bound always yields “`true`” whenever the left bound is less than or equal to the right bound.

Theorem 3.4.5. ***Disjunction of Left and Right Bounds***: *Consider an arbitrary left bound B_L and an arbitrary right bound B_R. Left bound B_L is composed of a limit* *and a value: $B_L = l_L v_L$ while right bound B_R is composed of a value and a limit: $B_R = v_R l_R$. Then:*

$$B_L \leq B_R \qquad \Rightarrow \qquad B_L + B_R = \texttt{true}$$

***Proof**: There are two cases to consider. In the first case, $B_L \leq B_R$ because the values $v_L < v_R$. In the second case, $B_L \leq B_R$ because the values are equal to a value v say, but the limits $l_L \leq l_R$. Consider a variable $x \in T$.*

In the first case, $v_L < v_R$. Since all values $t \in T$ are by our definition able to be totally ordered, then it is axiomatic that $(x \leq v_R) + (v_R \leq x) = \texttt{true}$ by the axiom of totality (Gemignani 1990). But $v_L < v_R$ so we may write:

$$\begin{aligned} x \leq v_R \ + \ v_R \leq x &= \texttt{true} \\ x \leq v_R \ + \ v_L \leq x &= \texttt{true} \\ x \geq v_L \ + \ x \leq v_R &= \texttt{true} \\ B_L \ + \ B_R &= \texttt{true} \end{aligned}$$

In the second case, inspection of Algorithm 3.4.1 (page 53) shows that the only way left limit l_L can be less than or equal to right limit l_R is when $l_L = [$ and $l_R =]$. Then:

$$\begin{aligned} B_L + B_R &= (x \geq v) + (x \leq v) \\ &= \texttt{true} \end{aligned}$$

3.4.7.1 Summary: Conjunction and Disjunction of Bounds

We now summarise the results for the conjunction and disjunction of bounds from this section in Table 3.7.

Description	Operation	Algorithm
Conjunction of left bounds	$B_{L_1} \cdot B_{L_2}$	$\textit{higher}\,(B_{L_1}, B_{L_2})$
Conjunction of right bounds	$B_{R_1} \cdot B_{R_2}$	$\textit{lower}\,(B_{R_1}, B_{R_2})$
Disjunction of left bounds	$B_{L_1} + B_{L_2}$	$\textit{lower}\,(B_{L_1}, B_{L_2})$
Disjunction of right bounds	$B_{R_1} + B_{R_2}$	$\textit{higher}\,(B_{R_1}, B_{R_2})$
Disjunction of left and right bounds	$B_L + B_R$	`true` if $B_L \leq B_R$

Table 3.7: **Summary of important results for conjunction and disjunction of bounds**.

3.5 Intervals

We now define the data type, *interval*, which will be central to our algebra of intervals. Our definition is a generalisation of an interval on the Real number line (Muñoz & Lester 2005). We first show informally in Example 3.5.1 how left and right bounds combine to construct an *interval.*

Example 3.5.1. *Consider the two Real number instances $a, b \in \mathbb{R}$ and a variable $x \in \mathbb{R}$. Using the four bounds defined above in Definition 3.4.1 we may form four real intervals which have the interpretation described in Table 3.8. Note that if*

Left Bound	Right Bound	Interval	Interpretation
$[a$	$b]$	$[a,b]$	$\{x : a \le x \le b\}$
$(a$	$b]$	$(a,b]$	$\{x : a < x \le b\}$
$[a$	$b)$	$[a,b)$	$\{x : a \le x < b\}$
$(a$	$b)$	(a,b)	$\{x : a < x < b\}$

Table 3.8: **Four intervals and their interpretation**.

$b < a$, the interval is empty. The special case of the interval $[a, a]$ denotes the point a.

Definition 3.5.1. ***Interval**: Consider our data type T for which we have a well defined total ordering. Consider an arbitrary left bound B_L and an arbitrary right bound B_R. An interval I_T over the domain of T is defined by the Boolean conjunction of the left and right bounds. That is:*

$$I_T = B_L \cdot B_R$$

But the left and right bounds can be written in terms of their values $t_L, t_R \in T$ plus an associated limit operator $\langle, \rangle$:

$$\begin{array}{rclll} B_L & = & \langle t_L & \textit{where} & \langle \in \{ (,[\} \\ B_R & = & t_R\rangle & \textit{where} & \rangle \in \{),] \} \end{array}$$

So we may write:

$$\begin{array}{rcl} I_T & = & B_L \cdot B_R \\ & = & \langle t_L \cdot t_R \rangle \end{array}$$

Example 3.5.2. *Figure 3.4 depicts some arbitrary intervals on the Real number line. In each case we write the interval depicted using the familiar notation of inclusive and exclusive real intervals and underneath write the equivalent expression using Boolean operators.*

Our interval definition imposes no extra conditions other than those already discussed. We insist only that B_L and B_R are left and right bounds respectively. In particular, we say nothing as to whether interval I_T actually encloses any instances $t \in T$.

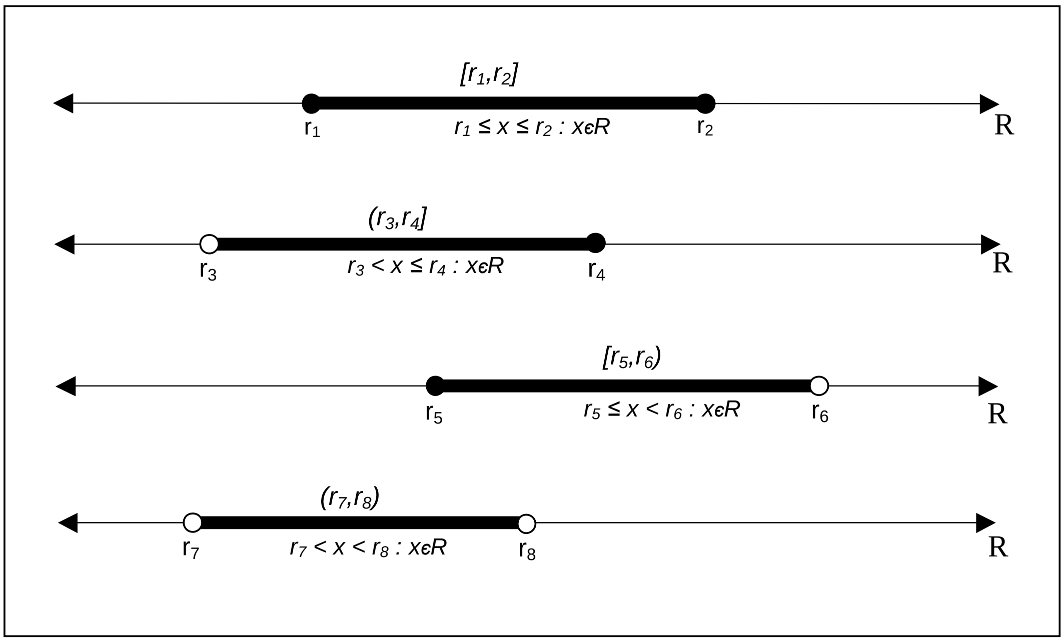


Figure 3.4: **Intervals on the Real number line**: Points r_1 to r_8 are arbitrary points on the Real number line. Using these points we construct some intervals on the Real number line. In each case we write the interval depicted using the familiar notation of inclusive and exclusive intervals and underneath write the equivalent expression using Boolean comparison operators.

3.5.1 Special Intervals

In Definition 3.5.1 (page 59) we defined the *interval* data type that will be central to our interval algebra. Before we can describe the algebra itself, we first define some special intervals which we then utilise in our description. Our goal is to define an algebra which is analogous to Boolean algebra. We therefore require some *identity elements*, analogous to the Boolean constants "`true`" and "`false`". These are the special intervals we describe in the remainder of this section and this foreshadows Section 3.10 (page 74).

3.5.1.1 The Infinite Interval

In Definitions 3.2.1 and 3.2.2 (page 47) we defined two special values, "$MINF$" and "$PINF$" which we use as constants to denote values *minus infinity* and *plus infinity* respectively. Then in Definitions 3.4.2 and 3.4.3 (page 53) we utilised these two constants to define an infinite left bound "MIB" and an infinite right bound "PIB". We now use these special bounds along with Definition 3.5.1 (page 59) to define the *infinite interval.*

Definition 3.5.2. ***Infinite Interval**: This is the interval defined by setting the left bound to MIB and the right bound to PIB. We use the symbol* **1** *to denote the* *infinite interval. Therefore we may write:*

$$\begin{aligned} MIB, PIB &\equiv [MINF, PINF] \\ &\equiv \mathbf{1} \end{aligned}$$

If we think of the infinite interval as an assertion, it asserts that the valid or legal range of values for this domain must lie between plus or minus infinity, an assertion that is *always true*. We employ the symbol "**1**" across all data types represented by T, in each case relying on context to convey the type of infinite interval to which we are referring.

Theorem 3.5.1. *The infinite interval* **1** *is equivalent*[2] *to the Boolean constant "*`true`*".*
Proof*: From Definition 3.5.2 above we may write the infinite interval as* $[MINF, PINF]$*. But this is equivalent to the Boolean expression:*

$$\{x : \ MINF \leq x \leq PINF\}$$

for some variable $x \in T$*. By Definitions 3.2.1 and 3.2.2 (page 47), this expression must always be true since all instances of* T *must fall (inclusively) between* $MINF$ *and* $PINF$*.*

3.5.1.2 The Null Interval

Definition 3.5.1 (page 59), our interval definition, makes no assumption as to the relative rank of bounds making up the interval. For example, if our data type is the Real numbers, we do not assume the left value is always less than or equal to the right value, as intuition might dictate.

Example 3.5.3. *Figure 3.5 depicts an arbitrary interval on the Real number line where the right bound* $r_2]$ *is approaching the left bound* $[r_1$*. Eventually the bounds enclose the single value* $r_1 = r_2$*. But there are no constraints on the values of* r_1 *and* r_2 *so, for example,* r_2 *may be less than* r_1*, giving rise to a null interval. The interval still exists, as do the bounds that comprise it. However, the interval encloses no instances of* $r \in \mathbb{R}$*.*

Definition 3.5.3. ***The Null Interval****: Consider an arbitrary interval* I *over our domain* T *consisting of a left bound* B_L *and a right bound* B_R*. That is,* $I = B_L \cdot B_R$*. Then* I *is the null interval if and only if* $B_R < B_L$*. We use the symbol* **0** *to denote the null interval. We may write:*

$$I = \mathbf{0} \quad \Leftrightarrow \quad B_R < B_L$$

[2]We mean "equivalent" in the sense that when an interval is considered as a Boolean assertion, the infinite interval behaves exactly like the truth value "`true`".

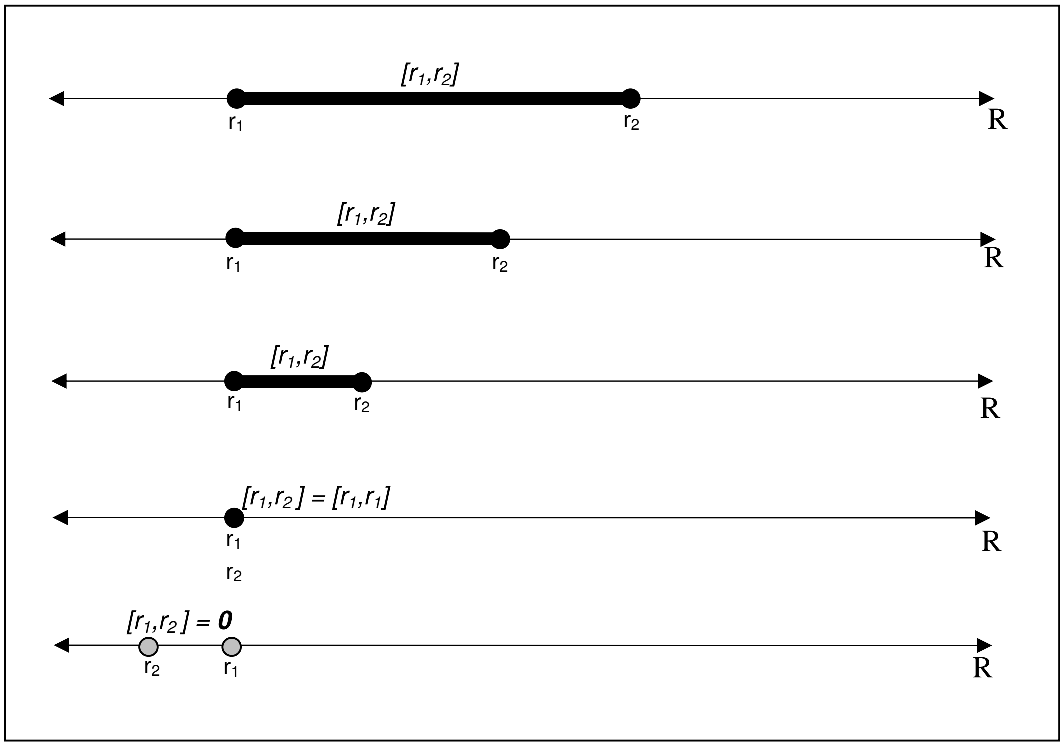


Figure 3.5: **Intervals on the Real number line**: We picture an arbitrary interval on the Real number line where the right bound $r_2]$ is approaching the left bound $[r_1$. In the first three number lines, $r_2 > r_1$. Eventually the bounds enclose the single value $r_1 = r_2$. In the final number line, $r_2 < r_1$ and the interval is *null.*

The null interval, as we have defined it above, never encloses any values and is analogous to the *empty set* which contains no members. If we think of the null interval as an assertion, it asserts that there is *no* valid or legal range of values enclosed by the interval, in this domain. We employ the symbol “**0**” across all data types, in each case relying on context to convey the type of null interval to which we are referring. Several consequences follow immediately from the above definition.

Theorem 3.5.2. *Consider the arbitrary values $t, t' \in T$ such that $t' > t$. Then the following is true of the intervals formed from values t and t':*

$$(t, t) = \mathbf{0} \tag{3.1}$$

$$[t, t) = \mathbf{0} \tag{3.2}$$

$$(t, t] = \mathbf{0} \tag{3.3}$$

$$\langle t', t \rangle = \mathbf{0} \tag{3.4}$$

***Proof**: The right bound in all the above is less than the left bound.*

In the case of the first three Equations 3.1 to 3.3, where the left and right values $t \in T$ are identical, the rank of the bound is determined solely by comparing the left limit operator with the right limit operator. This can be looked up in Table 3.2 (page 49) and in each case returns **1**, *indicating the left bound is greater than the* *right bound. Therefore by Definition 3.5.3 above (page 61), in each of the three cases the interval must be null.*

In the case of the last Equation 3.4, by Algorithm 3.4.1 (page 53), the right bound must be less than the left bound because $t' > t$, irrespective of the value of the left limit or the right limit. Therefore by Definition 3.5.3 above, the interval must be null.

Corollary 3.5.2.1. *There are no other forms of the null interval other than the cases depicted in Equations 3.1 to 3.4.*
***Proof**: By Definition 3.5.3, an interval is null if and only if its right bound is less than its left bound.*

When both left and right values *are the same there can only be four different ways of writing the limits (two left limits times two right limits). The only permutation missing from Equations 3.1 to 3.3 is "$[t, t]$" which is not null since the bounds are equal. Therefore, the three forms depicted in Equations 3.1 to 3.3 are the only null intervals when the values are the same.*

When the values are different, by Algorithm 3.4.1 (page 53), the limits are irrelevant and the rank of the bounds is determined solely by the rank of the values. Therefore, the only other case to consider is when $t' \leq t$. But in this case, the left bound must be less than or equal to the right bound, so the interval cannot be null. Therefore Equation 3.4 is the only interval form which is null when the values are different.

Corollary 3.5.2.2. *The null interval* **0** *is equivalent*[3] *to the Boolean constant "*`false`*".*
***Proof**: From Theorem 3.5.2 above, the interval (t, t) is always null, for all $t \in T$. So whenever the null interval occurs we might just as well substitute (t, t). But we can think of (t, t) as the following logical assertion about some variable $x \in T$:*

$$\{x : \; x > t \; \cdot \; x < t\}$$

where t is some constant. This assertion is always false. Similarly, we can substitute any of the other forms of the null interval depicted in Equations 3.1 to 3.4. All four cases are set out in Table 3.9 below. In each case the truth value of the assertion evaluates to `false`*.*

3.6 Conjunction of Intervals

In this section we define a binary operator *conjunction* which is analogous to Boolean conjunction. We first derive a theorem for this operation using Boolean algebra. We

[3]We mean "equivalent" in the sense that when an interval is considered as a Boolean assertion, the null interval behaves exactly like the truth value "`false`".

Null Interval	Assertion	Truth Value
(t, t)	$\{x : x > t \cdot x < t\}$	`false`
$[t, t)$	$\{x : x \geq t \cdot x < t\}$	`false`
$(t, t]$	$\{x : x > t \cdot x \leq t\}$	`false`
$\langle t', t\rangle$	$\{x : x \succ t' \cdot x \prec t \cdot t' > t\}$	`false`

Table 3.9: **Corollary 3.5.2.2**: Each form of the null interval may be rewritten as a logical assertion. In each case, the truth value of the assertion is seen to be `false`. The symbol "$\succ$" denotes either "$>$" or "$\geq$". Similarly, the symbol "$\prec$" denotes either "$<$" or "$\leq$".

then give an explicit algorithm for this operation.

Informally, the conjunction of two intervals is the interval formed by the *higher of left bounds* and *lower of right bounds*. Conjunction of intervals is illustrated in Figure 3.6.

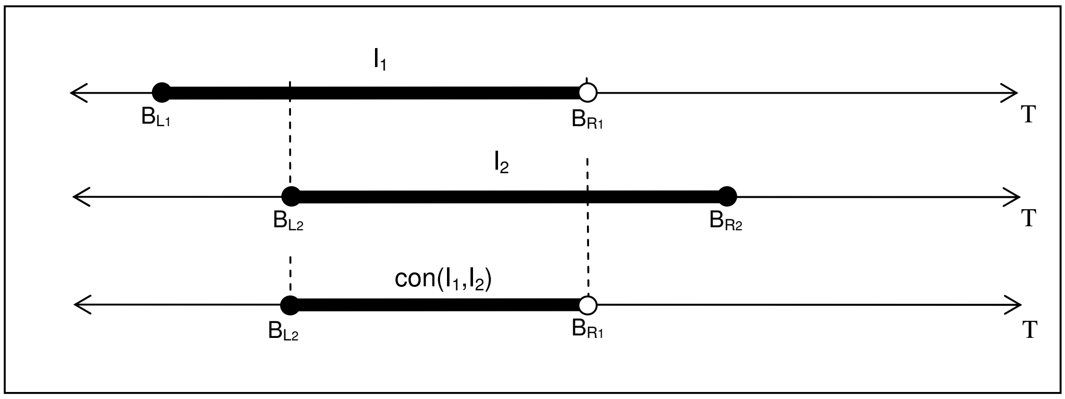


Figure 3.6: **The Boolean conjunction of two intervals** I_1 and I_2 is represented graphically by the intersection of the intervals. The conjunction $I_1 \cdot I_2$ returns another interval which can never be more expansive than either I_1 or I_2. If the intervals are disjoint, their intersection is null.

Theorem 3.6.1. ***Conjunction of Intervals**: The Boolean conjunction of interval I_1 with interval I_2 is given by:*

$$I_1 \cdot I_2 = higher(B_{L_1}, B_{L_2}) \cdot lower(B_{R_1}, B_{R_2})$$

***Proof**:*

$$\begin{aligned}
I_1 \cdot I_2 &= (B_{L_1} \cdot B_{R_1}) \cdot (B_{L_2} \cdot B_{R_2}) \\
&= B_{L_1} \cdot B_{R_1} \cdot B_{L_2} \cdot B_{R_2} \\
&= (B_{L_1} \cdot B_{L_2}) \cdot (B_{R_1} \cdot B_{R_2}) \\
&= higher(B_{L_1}, B_{L_2}) \cdot (B_{R_1} \cdot B_{R_2}) && \textit{by Thm 3.4.1} \\
&= higher(B_{L_1}, B_{L_2}) \cdot lower(B_{R_1}, B_{R_2}) && \textit{by Thm 3.4.2}
\end{aligned}$$

We now use the above theorem to define a conjunction algorithm for intervals. The following algorithm uses the functions "*lower*" and "*higher*" from Algorithms 3.4.2 and 3.4.3 (page 54).

Algorithm 3.6.1. ***Conjunction of Intervals****: The conjunction "con" of interval I_1 with interval I_2 is given by the following pseudo-code function:*

```
$con(I_1, I_2)$ return interval is
    return  $higher(B_{L_1}, B_{L_2}) \cdot lower(B_{R_1}, B_{R_2})$ ;
```

The conjunction of two intervals always returns another (possibly null) interval. That is, the binary operation conjunction is *closed* for intervals. Furthermore, the conjunction of two intervals can never result in an expanded interval; i.e., the resulting interval can never encompass more than the smaller of the two intervals.

3.7 Disjunction of Intervals

In this section we define a binary operator *disjunction* which is analogous to Boolean disjunction. We first define precisely what we mean by intervals that *overlap*, intervals that *touch* and intervals that are *disjoint*. We then derive a theorem for the disjunction operation using Boolean algebra.

Informally, the disjunction of two intervals is given by the following. If two intervals overlap or they touch, their disjunction is the interval formed by the *lower of left bounds* and *higher of right bounds*. This is illustrated in Figures 3.7(a) and 3.7(b). If the intervals are disjoint, the result is the interval *list* containing the two (disjoint) intervals. This is illustrated in Figure 3.7(c).

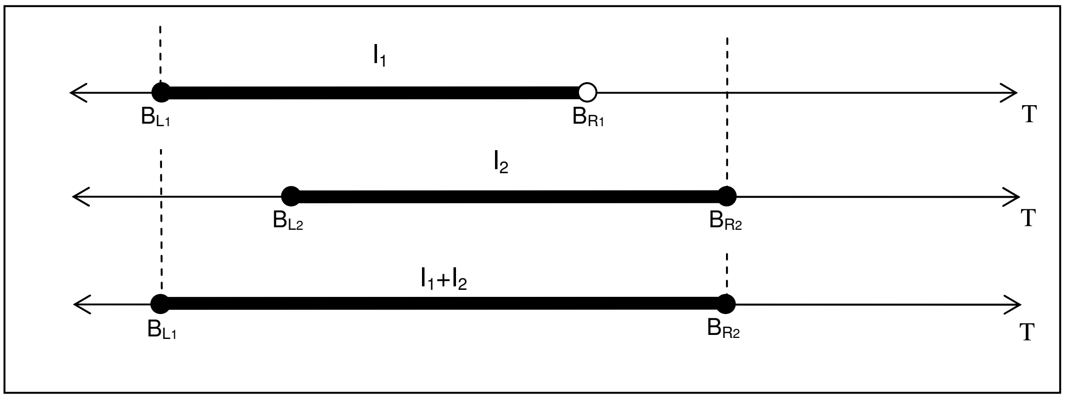
Intuitively, two intervals *overlap* when their intersection is non-null. Refer to Figure 3.7(a). Using the result of Theorem 3.5.2 and Theorem 3.6.1, this leads immediately to the following definition.

Definition 3.7.1. ***Interval Overlap****: Consider two arbitrary intervals I_1 and I_2 over T. These intervals overlap if and only if their intersection is non-null. That is:*

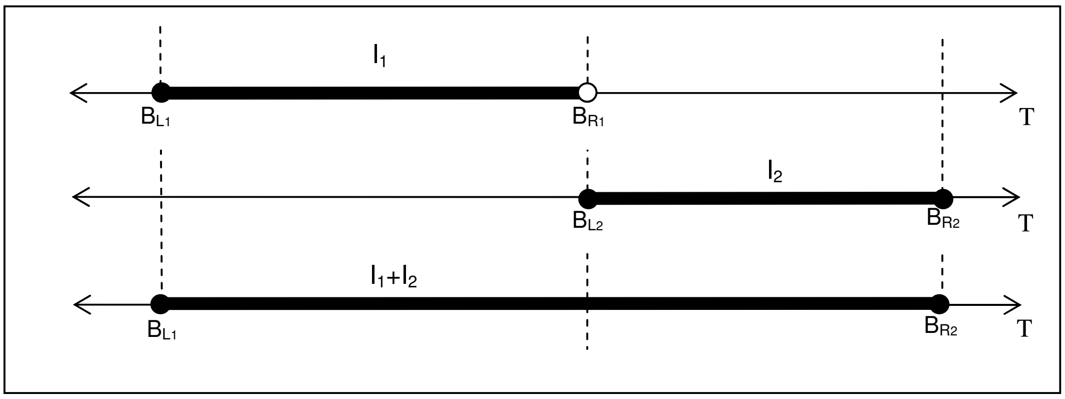
$$\begin{aligned} overlap(I_1, I_2) &\Leftrightarrow \neg\,(I_1 \cdot I_2 = \mathbf{0}) \\ &\Leftrightarrow \neg\,(higher(B_{L_1}, B_{L_2}) > lower(B_{R_1}, B_{R_2})) \\ &\Leftrightarrow higher(B_{L_1}, B_{L_2}) \le lower(B_{R_1}, B_{R_2}) \end{aligned}$$

Intuitively, two intervals *touch* when the right bound of one interval has the same value as the left bound of the other interval, but one limit is inclusive while the other is exclusive (or vice versa). Refer to Figure 3.7(b). This leads to the following definition.

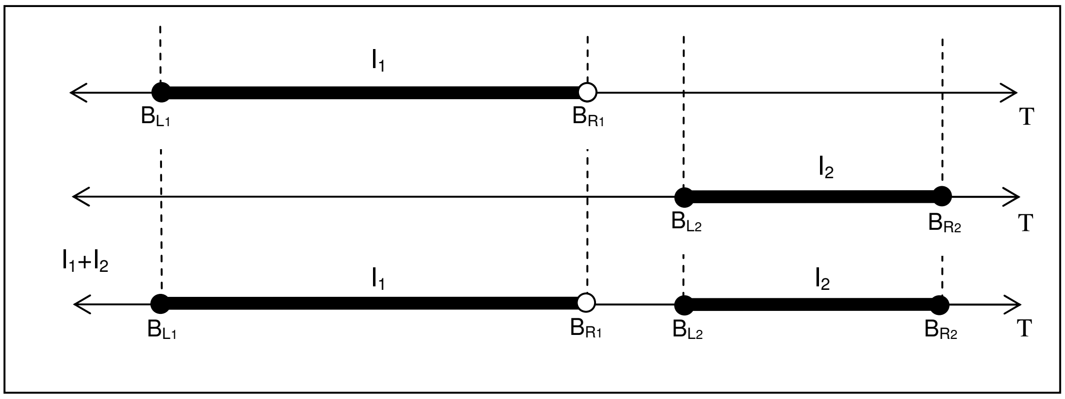
Definition 3.7.2. ***Interval Touch****: Consider two arbitrary intervals I_1 and I_2 over T comprising values $a_1, b_1, a_2, b_2 \in T$ and limits $\langle_1, \langle_2 \in \{\, (, [\, \}$ and $\rangle_1, \rangle_2 \in \{\,),]\, \}$ such*



(a) Disjunction of overlapping intervals



(b) Disjunction of touching intervals



(c) Disjunction of disjoint intervals

Figure 3.7: **The Boolean disjunction of two intervals** I_1 and I_2 is represented graphically by the union of the intervals. If the intervals overlap or touch, the disjunction is formed by lower of left bounds and the higher of right bounds. If the intervals are disjoint, the disjunction $I_1 + I_2$ returns an *interval list* comprised of the same two intervals.

that

$$
\begin{aligned}
I_1 &= B_{L_1} \cdot B_{R_1} \\
&= \langle_1 a_1 \ \cdot \ b_1\rangle_1 \\
I_2 &= B_{L_2} \cdot B_{R_2} \\
&= \langle_2 a_2 \ \cdot \ b_2\rangle_2
\end{aligned}
$$

Writing the negation of left limit $\langle$ as $\langle'$ and negation of right limit $\rangle$ as $\rangle'$, these intervals touch *if and only if the following is true:*

$$touch(I_1, I_2) \quad \Leftrightarrow \quad \big(b_1 = a_2 \;\cdot\; \rangle_1 = \langle'_2\big) \;+\; \big(a_1 = b_2 \;\cdot\; \langle_1 = \rangle'_2\big)$$

We now define disjoint intervals as intervals that neither overlap nor touch.

Definition 3.7.3. ***Disjoint Intervals****: Consider two arbitrary intervals I_1 and I_2 over T. These intervals are disjoint if and only if they neither overlap nor touch. That is:*

$$disjoint(I_1, I_2) \quad \Leftrightarrow \quad \neg overlap(I_1, I_2) \;\cdot\; \neg touch(I_1, I_2)$$

Now we utilise Definition 3.7.3 above to derive a theorem for the disjunction of two intervals.

Theorem 3.7.1. ***Disjunction of Intervals****: The Boolean disjunction of interval I_1 with interval I_2 is given by the following.*

- *If the intervals overlap or touch then:*

$$I_1 + I_2 = lower(B_{L_1}, B_{L_2}) \;\cdot\; higher(B_{R_1}, B_{R_2})$$

- *If the intervals are disjoint then:*

$$I_1 + I_2 = \{I_1, I_2\}$$

Proof*: The proof of the second case above when the intervals are disjoint is trivial. In the first case, we consider first when the intervals overlap, then when the intervals touch.*

Consider Figure 3.7(a) (page 66), which depicts overlapping intervals. We have:

$$\begin{aligned} I_1 + I_2 &= (B_{L_1} \cdot B_{R_1}) \;+\; (B_{L_2} \cdot B_{R_2}) \\ &= (B_{L_1} + B_{L_2}) \cdot (B_{L_1} + B_{R_2}) \cdot (B_{R_1} + B_{L_2}) \cdot (B_{R_1} + B_{R_2}) \end{aligned} \tag{3.5}$$

Looking at the first term of Equation 3.5 above:

$$(B_{L_1} + B_{L_2}) = lower(B_{L_1}, B_{L_2}) \qquad \textit{by Theorem 3.4.3}$$

Similarly, looking at the last term of Equation 3.5 above:

$$(B_{R_1} + B_{R_2}) = higher\,(B_{R_1}, B_{R_2}) \qquad \textit{by Theorem 3.4.4}$$

So we may write $I_1 + I_2$ as:

$$lower\,(B_{L_1}, B_{L_2}) \cdot higher\,(B_{R_1}, B_{R_2}) \cdot (B_{L_1} + B_{R_2}) \cdot (B_{R_1} + B_{L_2}) \tag{3.6}$$

But the remaining terms in Equation 3.6 above, $(B_{L_1} + B_{R_2})$ and $(B_{R_1} + B_{L_2})$ are both equal to "`true`" by Theorem 3.4.5 since in both cases the left bound is less than or equal to the right bound (otherwise there would be no overlap). So:

$$\begin{aligned} I_1 + I_2 &= lower\,(B_{L_1}, B_{L_2}) \cdot higher\,(B_{R_1}, B_{R_2}) \cdot \texttt{true} \cdot \texttt{true} \\ &= lower\,(B_{L_1}, B_{L_2}) \cdot higher\,(B_{R_1}, B_{R_2}) \end{aligned}$$

Now consider Figure 3.7(b) (page 66), which depicts touching intervals. Referring to Equation 3.6 above, the term $(B_{L_1} + B_{R_2})$ is again equal to "`true`" by Theorem 3.4.5. At the touching bounds, the values are equal to a value v say, but one limit is inclusive while the other is exclusive (by Definition 3.7.2). Suppose the touching right bound is inclusive; i.e., $B_{R_1} = v]$. Then the touching left bound must be $B_{L_2} = (v$. Then the term $(B_{R_1} + B_{L_2})$ in Equation 3.6 above is given by:

$$\begin{aligned} (B_{R_1} + B_{L_2}) &= B_{R_1} = v] \;+\; (v \\ &= \{x:\; x \le v \;+\; x > v\} \\ &= \texttt{true} \end{aligned}$$

Therefore, Equation 3.6 again reduces to:

$$\begin{aligned} I_1 + I_2 &= lower\,(B_{L_1}, B_{L_2}) \cdot higher\,(B_{R_1}, B_{R_2}) \cdot \texttt{true} \cdot \texttt{true} \\ &= lower\,(B_{L_1}, B_{L_2}) \cdot higher\,(B_{R_1}, B_{R_2}) \end{aligned}$$

The intervals of Figure 3.7(a) depicts interval I_1 preceding interval I_2. It is also possible that interval I_2 precedes interval I_1. In this case, the proof proceeds in an identical fashion to the above.

The disjunction of two intervals produces either another interval (if the two intervals overlap or touch) or exactly the two original intervals (if the two intervals are disjoint). Therefore, the binary operation disjunction is *not closed* for intervals, since a set or collection of intervals results in the case where the intervals are disjoint. This result is the motivation for a new data type, the *interval list*, which is a disjunction of disjoint intervals. We define precisely what we mean by *interval list* in Section 3.8 below (page 69).

We complete this section by using Definitions 3.7.1, 3.7.2 and 3.7.3 to define algorithms for interval *overlap, touch* and *disjointness.*

Algorithm 3.7.1. ***Overlap of Intervals***: *The Boolean function "overlap" of interval $I_1 = B_{L_1} \cdot B_{R_1}$ with interval $I_2 = B_{L_2} \cdot B_{R_2}$ is given by the following pseudo-code*

function:

```
$overlap(I_1, I_2)$ return boolean is
    return $higher(B_{L_1}, B_{L_2}) \leq lower(B_{R_1}, B_{R_2})$;
```

Algorithm 3.7.2. ***Touch of Intervals***: *Consider interval I_1 comprising left bound l_1a_1 and right bound b_1r_1 where l_1, r_1 are respectively the left and right limits and a_1, b_1 are respectively the left and right values. Similarly, interval I_2 comprises left bound l_2a_2 and right bound b_2r_2 where l_2, r_2 are respectively the left and right limits and a_2, b_2 are respectively the left and right values. Then the Boolean function "touch" of interval $I_1 = l_1a_1 \cdot b_1r_1$ with interval $I_2 = l_2a_2 \cdot b_2r_2$ is given by the following pseudo-code function. This function calls the negation of limits Algorithm 3.3.2 (page 50):*

```
$touch(I_1, I_2)$ return boolean is
    return $(b_1 = a_2$ and $r_1 = neg(l_2))$ or $(a_1 = b_2$ and $l_1 = neg(r_2))$;
```

Algorithm 3.7.3. ***Disjointness of Intervals***: *The Boolean function "disjoint" of interval $I_1 = B_{L_1} \cdot B_{R_1}$ with interval $I_2 = B_{L_2} \cdot B_{R_2}$ is given by the following pseudo-code function:*

```
$disjoint(I_1, I_2)$ return boolean is
    return not $overlap(I_1, I_2)$ and not $touch(I_1, I_2)$;
```

3.8 Interval Lists

We now introduce the *interval list*, which is simply a collection of intervals that do not intersect. We first describe informally what we mean by the term interval list, looking at it first as a logical assertion and then as a helpful data structure. We impose the special condition that our interval lists be composed of *disjoint* intervals. We then provide a precise definition.

It will often be convenient to picture a list of several intervals at once, with each interval defining a legal range of values, all with respect to a particular variable.

Example 3.8.1. *Referring to Figure 3.8, we have depicted two interval lists L_1 and L_2. Interval list L_1 is composed of the three disjoint intervals I_1, I_2 and I_3 while interval list L_2 is composed of the two disjoint intervals J_1 and J_2. The interpretation of the interval lists is straightforward. Interval list L_1 asserts that the valid or legal range of values (in this example the values are Real numbers) must lie between the bounds comprising I_1 or between the bounds comprising I_2 or between the bounds comprising I_3. Interval list L_2 asserts that the valid or legal range of values must lie between the bounds comprising J_1 or J_2.*

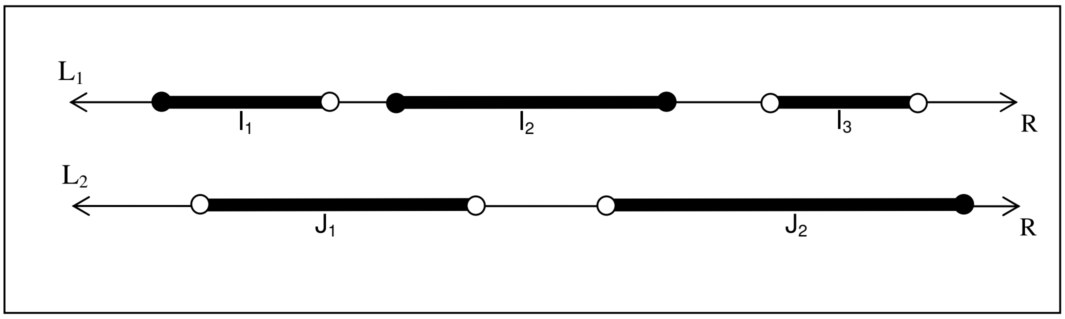


Figure 3.8: **Two interval lists**: Interval list L_1 is composed of the three disjoint intervals I_1, I_2 and I_3 while interval list L_2 is composed of the two disjoint intervals J_1 and J_2.

In Section 3.5 (page 58), we emphasized that an interval is an assertion about the legal or valid range of values a variable may assume. Logically, an interval list is a *disjunction* of such assertions; i.e., a list of intervals connected by the Boolean "*or*" operator.

Later in this chapter we will use both the interval and interval list as helpful data structures. As a data structure, an interval list is a *collection* or *set* of intervals in the sense that many programming languages use these terms (Doke et al. 2003).

Whenever we employ such a list of intervals, we will insist that each is disjoint; i.e., no interval overlaps or touches any other interval. However, we do not insist interval lists are in any sense an *ordered* collection of intervals. Indeed we make no attempt to define an ordering for intervals in this context[4].

3.8.1 Notation for Interval Lists

In order to conveniently represent interval lists, we borrow from the notation of summation algebra (Poole 2005) and employ the "$\sum$" symbol to denote Boolean disjunction and the "$\prod$" symbol to denote Boolean conjunction (Maurer 2004). The following definitions make this explicit.

Definition 3.8.1. ***Boolean Sum**: Consider a set of n Boolean assertions $A_1, A_2, \cdots, A_n$. We denote the disjunction of these assertions by:*

$$\sum_{i=1}^{n} A_i \equiv A_1 + A_2 + \cdots + A_n$$

Definition 3.8.2. ***Boolean Product**: Consider a set of n Boolean assertions*

[4]In fact it is possible to define a deterministic ordering for intervals. We have implemented such an ordering based on consideration first of the rank of the intervals' left bounds. If these are equal we then consider the rank of the intervals' right bounds. However, we do not utilise interval ordering for the disjunction, conjunction and negation operations we define in this chapter.

$A_1, A_2, \cdots, A_n$. *We denote the conjunction of these assertions by:*

$$\prod_{i=1}^{n} A_i \equiv A_1 \cdot A_2 \cdot \; \cdots \; \cdot A_n$$

In order to emphasize an interval list L is a disjunction of *disjoint* intervals, when it is convenient to do so, we will write the disjunction simply as a list enclosed by braces in the following manner:

$$\begin{aligned} L &= I_1 + I_2 + \; \cdots \; + I_n \\ &= \{I_1, I_2, \; \cdots \; , I_n\} \end{aligned}$$

3.8.2 Definition of Interval List

We can now precisely define what we mean by an interval list.

Definition 3.8.3. ***Interval List***: *Consider a set of n disjoint intervals $I_1, I_2, \cdots, I_n$ over the domain T. Then an interval list L is the Boolean disjunction of these intervals. That is:*

$$\begin{aligned} L &= \sum_{i=1}^{n} I_i \\ &= I_1 + I_2 + \; \cdots \; + I_n \\ &= \{I_1, I_2, \; \cdots \; , I_n\} \end{aligned}$$

3.8.3 Interval Disjunction Algorithm

We can now give an algorithm for the disjunction of two intervals. The data type returned by the function is always an interval list consisting of either one interval (when the intervals overlap or touch) or two intervals (when the intervals are disjoint). This algorithm makes use of Algorithm 3.7.3, the "*disjoint*" function (page 69). Interval disjunction is depicted in Figure 3.7 (page 66).

Algorithm 3.8.1. ***Disjunction of Intervals***: *The disjunction "dis" of interval I_1 with interval I_2 is given by the following pseudo-code function:*

```
dis($I_1, I_2$) return interval_list is
    $L \leftarrow \{\}$;
    if disjoint($I_1, I_2$) then
        $L \leftarrow \{I_1 \, , \, I_2\}$;
    else
        $L \leftarrow \{lower(B_{L_1}, B_{L_2}) \cdot \; higher(B_{R_1}, B_{R_2})\}$
    endif;
    return L;
```

3.9 Negation of Intervals

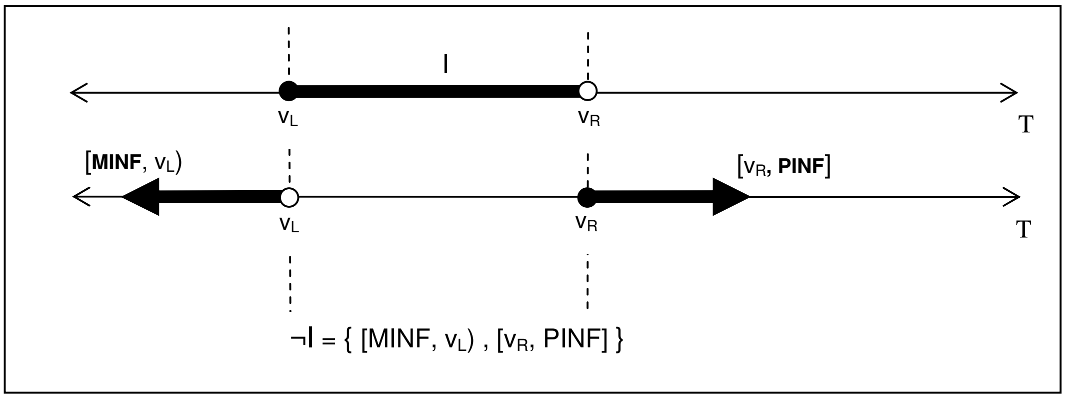


Figure 3.9: **Interval negation**: Interval $I = [v_L, v_R)$. Negating interval I gives everything *except* I. This results in an interval *list* consisting of exactly two intervals. The original values v_L and v_R are unchanged; only their limits are negated.

The negation of interval I encloses every value *except* the values enclosed by interval I. Consider Figure 3.9 which illustrates the negation of an interval. Considering an interval as the Boolean conjunction of a left bound with a right bound, negating an interval must negate this conjunction. This leads directly to the Theorem 3.9.1 below.

Theorem 3.9.1. ***Negation of Interval**: Consider a non-null interval I over domain T. We can decompose this interval into its left and right bounds B_L and B_R. Then each bound can be decomposed further into a value and an associated limit:*

$$\begin{aligned} I &= B_L \cdot B_R \\ &= \langle v_L \cdot v_R \rangle \end{aligned}$$

Let the negation of left limit $\langle$ be written as $\langle'$ and the negation of right limit $\rangle$ be written as $\rangle'$, determined by looking up Table 3.3 (page 50). We make use of the two constants $MINF$ and $PINF$ from Section 3.2.5 (page 47). Then the negation $\neg I$ of interval I is given by the following interval list composed of exactly two intervals:

$$\neg I = \{\ [MINF \cdot v_L\langle'\ ,\ \rangle' v_R \cdot PINF]\ \}$$

***Proof**: For simplicity we will assume the left and right limits are both inclusive. Considering an interval as the Boolean conjunction of a left bound with a right* *bound and negating, we get:*

$$\begin{aligned}
\neg I &= \neg\,(B_L \;\cdot\; B_R) \\
&= \neg\;[v_L \;\;+\;\; \neg\, v_R] \\
&= v_L) \;+\; (v_R && \textit{by Def 3.4.4} \\
&= \{x :\; (x < v_L) \;+\; (v_R < x)\} \\
&= \{x :\; (MINF \le x < v_L) \;+\; (v_R < x \le PINF)\} && \textit{by Defs 3.2.1, 3.2.2} \\
&= [MINF, v_L) \;+\; (v_R, PINF] && \textit{by Def 3.5.1} \\
&= \{\,[MINF, v_L)\;,\;(v_R, PINF]\,\} && \textit{by Def 3.8.3}
\end{aligned}$$

In the above proof we assumed the left and right limits were both inclusive, but an analogous proof can be written for any combination of limits.

With the exception of the two special cases when the left bound is $MINF$ or the right bound is $PINF$, the negation of an interval returns an interval *list*. Therefore, the unary operation negation is *not closed* for intervals.

We now derive the results for the negation of the two special intervals **0** (the null interval) and **1** (the infinite interval).

Theorem 3.9.2. ***Negating the Null Interval**: The negation of the null interval* **0** *is the infinite interval* **1***.*
***Proof**:*

$$\begin{aligned}
0 &= (t, t) \qquad \forall t \in T && \textit{by Thm 3.5.2} \\
&= \{x :\; x > t \;\cdot\; x < t\}
\end{aligned}$$

Negating the above expression, we may write:

$$\begin{aligned}
\neg\mathbf{0} &= \{x :\; \neg\,(x > t \;\cdot\; x < t)\} \\
&= \{x :\; \neg\,(x > t) + \neg\,(x < t)\} \\
&= \{x :\; (x \le t) + (x \ge t)\} \\
&= \mathbf{1} && \textit{by Thm 3.4.5}
\end{aligned}$$

Theorem 3.9.3. ***Negating the Infinite Interval**: The negation of the infinite interval* **1** *is the null interval* **0***.*

Proof:

$$\begin{aligned}
\mathbf{1} &= MIB \cdot PIB && \textit{by Defs 3.4.2, 3.4.3}\\
\neg\mathbf{1} &= \neg\,(MIB \cdot PIB)\\
&= \neg MIB + \neg PIB\\
&= \{x :\ \neg\,(x \geq MINF) + \neg\,(x \leq PINF)\,\}\\
&= \{x :\ (x < MINF) + (x > PINF)\}\\
&= \{x :\ \mathbf{0} + \mathbf{0}\}\\
&= \mathbf{0}
\end{aligned}$$

We now use Theorem 3.9.1 and Theorems 3.9.2 and 3.9.3 to construct an algorithm for interval negation.

Algorithm 3.9.1. ***Negation of Interval***: *In the following, we write the negation of left limit $\langle$ as $\langle'$ and the negation of right limit $\rangle$ as $\rangle'$. Then the negation "neg" of interval $I = \langle v_L, v_R\rangle$ is given by the following pseudo-code function:*

```
neg(I) return interval_list is
    $L \leftarrow \{\}$;
    if $I = \mathbf{0}$ then
        $L \leftarrow \{\mathbf{1}\}$
    elsif $I = \mathbf{1}$ then
        $L \leftarrow \{\mathbf{0}\}$
    else
        $L \leftarrow \{\,[MINF, v_L\langle'\ \ ,\ \ \rangle' v_R, PINF]\,\}$
    endif;
    return $L$;
```

3.10 Identity Elements For Intervals

In this section we use the definitions of the null and infinite intervals from Sections 3.5.1 (page 60) and the theorems for the conjunction, disjunction and negation of intervals to derive two *identity elements* for intervals. These are analogous to the Boolean constants "`true`" and "`false`".

3.10.1 Identity Element for Conjunction of Intervals

We now show how the infinite interval **1** from Definition 3.5.2 (page 60) acts as the identity element when we apply the binary operator "·" (conjunction) to intervals.

Theorem 3.10.1. *The infinite interval* **1** *is the identity element when we apply the binary operator "·" (conjunction) to intervals.*
Proof: *Consider an arbitrary interval I over our domain T and its conjunction with infinite interval* **1**. *We may write:*

$$\begin{array}{rcll}
I & = & B_L \cdot B_R & \\
\mathbf{1} & = & MIB \cdot PIB & \textit{by Def 3.5.2} \\
\textit{Therefore:} & & & \\
I \cdot \mathbf{1} & = & higher(B_L, MIB) \cdot lower(B_R, PIB) & \textit{by Thm 3.6.1} \\
 & = & B_L \cdot B_R & \textit{by Defs 3.4.2, 3.4.3} \\
 & = & I & \\
\textit{Similarly:} & & & \\
\mathbf{1} \cdot I & = & higher(MIB, B_L) \cdot lower(PIB, B_R) & \\
 & = & B_L \cdot B_R & \\
 & = & I &
\end{array}$$

Since I was an arbitrary interval, **1** *is therefore the identity element when we apply the binary operator "·" (conjunction) to intervals.*

3.10.2 Identity Element for Disjunction of Intervals

We now show how the null interval **0** from Definition 3.5.3 (page 61) acts as the identity element when we apply the binary operator "+" (disjunction) to intervals.

Theorem 3.10.2. *The null interval* **0** *is the identity element when we apply the binary operator "+" (disjunction) to intervals.*
Proof: *Consider an arbitrary interval I over our domain T and its disjunction with null interval* **0**. *Since the null interval cannot enclose any values, we know a priori that there can be no overlap with I. So we may write:*

$$\begin{array}{rcll}
I + \mathbf{0} & = & \{I\ ,\ \mathbf{0}\} & \textit{by Thm 3.7.1} \\
 & = & I\ +\ \texttt{false} & \textit{by Cor 3.5.2.2} \\
 & = & I & \\
\textit{Similarly:} & & & \\
\mathbf{0} + I & = & \{\mathbf{0}\ ,\ I\} & \\
 & = & \texttt{false}\ +\ I & \\
 & = & I &
\end{array}$$

Since I was an arbitrary interval, **0** *is therefore the identity element when we apply the binary operator "+" (disjunction) to intervals.*

3.10.3 Informal Examples: Negation When a Bound is Infinite

We now informally consider some examples of negating intervals where one or both of the bounds are infinite. Our purpose is to show that the algorithm we have stated for the negation of an interval is sound and that the way we propose to treat infinite values and bounds is sound.

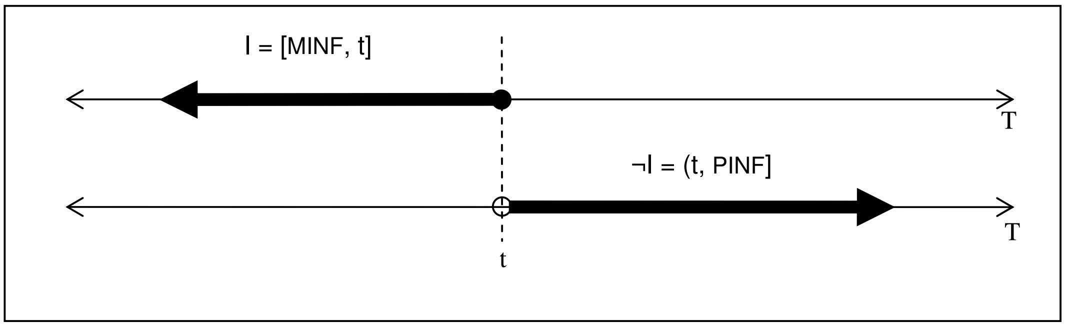


Figure 3.10: **Negation when one bound is infinite**: Interval I encompasses the entire domain T from $MINF$ (minus infinity) to $t \in T$. Its negation $\neg I$ therefore encompasses the complement of this interval. Refer to **Example 3.10.1**.

Example 3.10.1. *This example is illustrated in Figure 3.10. Consider an interval I comprising a left bound which is the infinite left bound MIB and an inclusive right bound given by $B_R = t]$. We wish to find the negation of I. We may write:*

$$\begin{array}{rcll} I & = & MIB, B_R & \\ & = & [MINF,\ t] & \textit{by Def 3.4.2} \\ \textit{Therefore:} & & & \\ \neg I & = & \neg\,([MINF,\ t]) & \\ & = & \{\,[MINF,\ MINF)\ ,\ (t,\ PINF]\,\} & \textit{by Thm 3.9.1} \\ & = & \{\,\mathbf{0}\ ,\ (t,\ PINF]\,\} & \textit{by Thm 3.5.2} \\ & = & \{\,(t,\ PINF]\,\} & \textit{by Thm 3.10.1} \end{array}$$

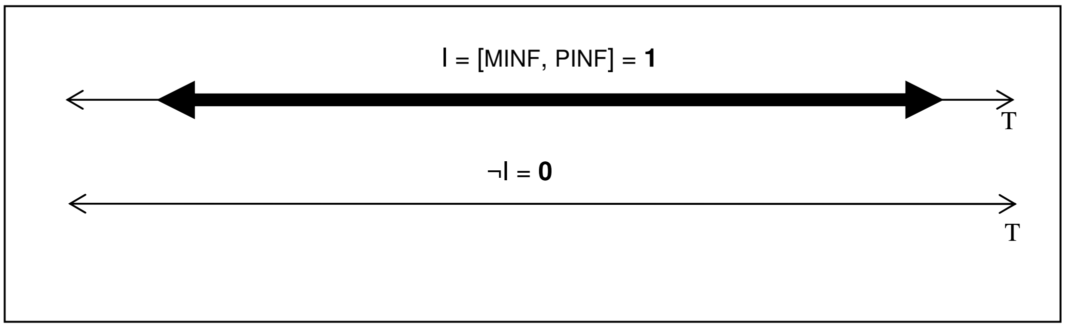


Figure 3.11: **Negation when both bounds are infinite**: Interval I occupies the entire domain T. Its negation $\neg I$ is the null interval list **0**. Refer to **Example 3.10.2**.

Example 3.10.2. *In this example we consider the extreme case where our interval I encompasses all values in the domain T. That is, I is the infinite interval. This example is illustrated in Figure 3.11. We may write:*

$$\begin{aligned}
I &= \mathbf{1} \\
&= MIB,\ PIB \\
&= [MINF,\ PINF] && \textit{by Def 3.5.2} \\
\textit{Therefore:} \\
\neg I &= \neg([MINF,\ PINF]) \\
&= \{\,[MINF,\ MINF)\ ,\ (PINF,\ PINF]\,\} && \textit{by Thm 3.9.1} \\
&= \{\,\mathbf{0}\ ,\ \mathbf{0}\} && \textit{by Thm 3.5.2} \\
&= \{\,\mathbf{0}\} && \textit{by Thm 3.10.1}
\end{aligned}$$

3.11 Interval Subsumption and Implication

In this section we define two further Boolean functions, *sub* (subsumes) and *imp* (implies) which operate on our interval data type. Our objective is to define a function analogous to the Boolean "implies". This will enhance the use of our interval algebra as a reasoning engine.

3.11.1 Interval Subsumption

We now consider subsumption of intervals. Consider Figure 3.12. We firstly define precisely what we mean by subsumption of intervals and then derive a theorem which follows from the definition.

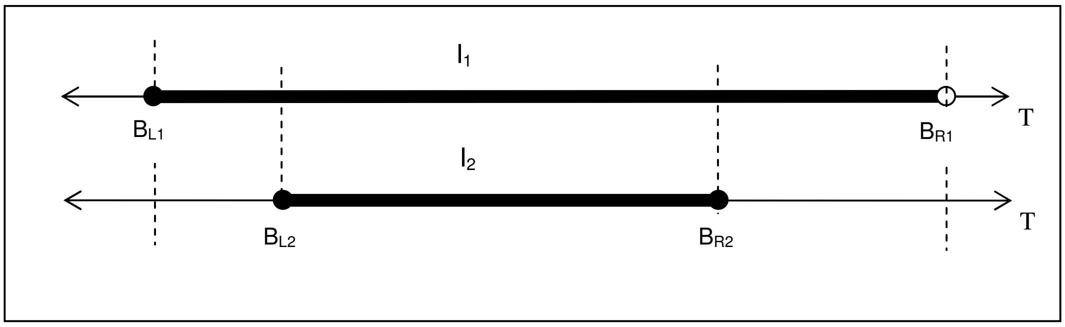


Figure 3.12: **Subsumption of intervals**: Interval I_1 by definition *subsumes* I_2 whenever its left bound B_{L_1} is less than or equal to B_{L_2} *and* its right bound B_{R_1} is greater than or equal to B_{R_2}.

Definition 3.11.1. ***Interval Subsumption***: *Consider two arbitrary intervals I_1 and I_2. Interval I_1 consists of left bound B_{L_1} and right bound B_{R_1}. Similarly, interval I_2 consists of left bound B_{L_2} and right bound B_{R_2}. Then I_1 subsumes I_2 if and only if $B_{L_1} \leq B_{L_2}$ and $B_{R_1} \geq B_{R_2}$. That is:*

$$I_1 \textit{ subsumes } I_2 \quad \Leftrightarrow \quad (B_{L_1} \leq B_{L_2}) \cdot (B_{R_1} \geq B_{R_2})$$

We now give an algorithm for the Boolean function "*sub*" (subsumes). This definition implicitly utilises the "$compare_{bound}$" function of Algorithm 3.4.1 (page 53).

Algorithm 3.11.1. ***Interval Subsumption***: *The Boolean function "sub" (subsumes) is given by the following pseudo-code function:*

```
$sub(I_1, I_2)$ return boolean is
    return $(B_{L_1} \leq B_{L_2})$ and $(B_{R_1} \geq B_{R_2})$;
```

We now prove a theorem which follows from Definition 3.11.1 above.

Theorem 3.11.1. ***Conjunction of Subsumed Intervals***: *Given intervals I_1 and I_2 as defined above in Definition 3.11.1 and given that I_1 subsumes I_2, then:*

$$I_1 \cdot I_2 = I_2$$

Proof: *From the conjunction algorithm of Theorem 3.6.1 (page 64) we have that*

$$I_1 \cdot I_2 = higher(B_{L_1}, B_{L_2}) \cdot lower(B_{R_1}, B_{R_2})$$

But using the subsumption definition above, $higher(B_{L_1}, B_{L_2}) = B_{L_2}$ while $lower(B_{R_1}, B_{R_2}) = B_{R_2}$. Therefore:

$$\begin{aligned} I_1 \cdot I_2 &= B_{L_2} \cdot B_{R_2} \\ &= I_2 \end{aligned}$$

3.11.2 Interval Implication

We now consider implication with intervals. We use the term "implies" precisely to mean Boolean implication and employ the symbol "→". We take as axiomatic that for any propositions P and Q:

$$P \to Q \equiv \neg P + Q$$

Consider again Figure 3.12. We firstly show that if interval I_1 subsumes interval I_2 then I_2 implies I_1.

Theorem 3.11.2. ***Interval Implication***: *Consider an interval $I_1 = B_{L_1} \cdot B_{R_1}$ which subsumes interval $I_2 = B_{L_2} \cdot B_{R_2}$. Then $I_2 \to I_1$.*

Proof *(by contradiction): Assume I_1 subsumes I_2 but that I_2 does not imply I_1. That* *is:*

$$\begin{aligned}\neg(I_2 \to I_1) &= \neg(\neg I_2 + I_1)\\ &= I_2 \cdot \neg I_1\\ &= B_{L_2} \cdot B_{R_2} \cdot \neg(B_{L_1} \cdot B_{R_1})\\ &= B_{L_2} \cdot B_{R_2} \cdot (\neg B_{L_1} + \neg B_{R_1})\\ &= (B_{L_2} \cdot B_{R_2} \cdot \neg B_{L_1}) + (B_{L_2} \cdot B_{R_2} \cdot \neg B_{R_1})\\ &= (B_{L_2} \cdot \neg B_{L_1} \cdot B_{R_2}) + (B_{R_2} \cdot \neg B_{R_1} \cdot B_{L_2})\end{aligned}$$

Consider the first term $(B_{L_2} \cdot \neg B_{L_1} \cdot B_{R_2})$*. Since by definition* $B_{L_1} \le B_{L_2}$*, it must be false that* $B_{L_2} \cdot \neg B_{L_1}$*. Similarly, considering the second term* $(B_{R_2} \cdot \neg B_{R_1} \cdot B_{L_2})$*, it must be false that* $B_{R_2} \cdot \neg B_{R_1}$*. Therefore:*

$$\begin{aligned}\neg(I_2 \to I_1) &= \texttt{false} + \texttt{false}\\ &= \texttt{false}\end{aligned}$$

Therefore it cannot be the case that $\neg(I_2 \to I_1)$ *and we therefore conclude that* $(I_2 \to I_1)$*.*

We now give an algorithm for the Boolean function "*imp*" (implies). This algorithm utilises the subsumption function of Algorithm 3.11.1.

Algorithm 3.11.2. ***Interval Implication****: The Boolean function "imp" (implies) is given by the following pseudo-code function:*

```
imp($I_1, I_2$) return boolean is
    return sub($I_2, I_1$);
```

3.12 Disjunction of Interval Lists

In this section we extend the definition of interval disjunction from Sections 3.7 (page 65) to apply to interval *lists*. We are led to this new definition by noting that an interval list (as we have defined it in Definition 3.8.3, page 71) is nothing more than a Boolean disjunction of intervals. We make use of the summation notation from Section 3.8.1 (page 70). The disjunction of interval lists on the Real number line is depicted in Figure 3.13 as the union of the two lists.

Theorem 3.12.1. ***Disjunction of Interval Lists****: Consider interval lists* $L_1 = \{I_1, I_2, \cdots, I_n\}$ *and* $L_2 = \{J_1, J_2, \cdots, J_m\}$*. Then the disjunction of these lists* $L_1 + L_2$ *is given by:*

$$L_1 + L_2 = \sum_{j=1}^{m} \sum_{i=1}^{n} \left(I_i + J_j\right)$$

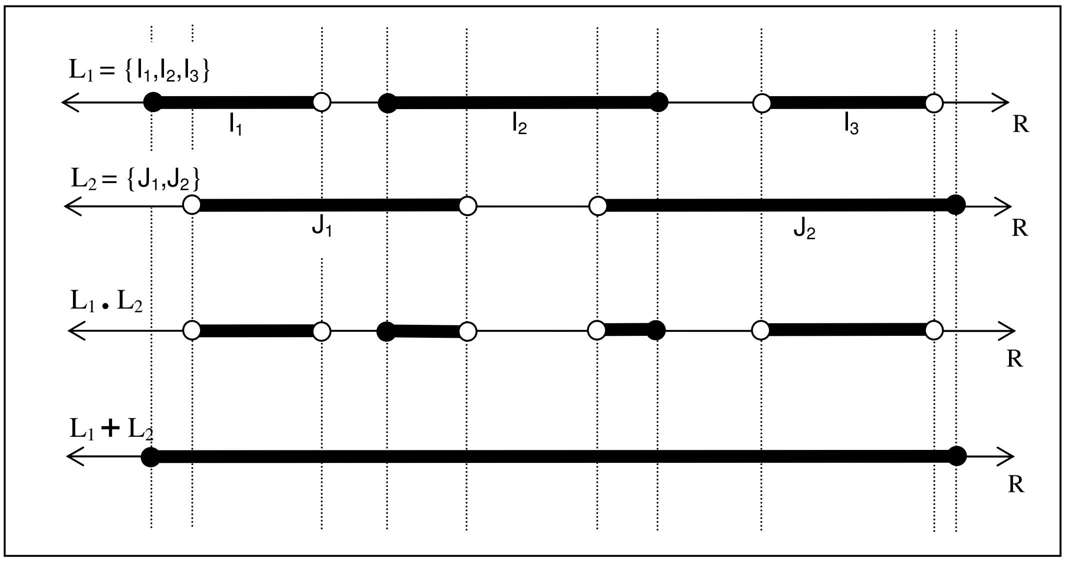


Figure 3.13: **Interval list conjunction and disjunction**: We may represent the conjunction of two interval lists L_1 and L_2 as the *intersection* of the two lists. Similarly, the disjunction of the two lists is the *union* of the two lists.

Proof:

$$
\begin{aligned}
L_1 + L_2 &= \sum_{i=1}^{n} I_i \; + \; \sum_{j=1}^{m} J_j \\
&= \quad (I_1 + I_2 + \cdots + I_n) + J_1 \\
&\quad + (I_1 + I_2 + \cdots + I_n) + J_2 \\
&\quad \vdots \\
&\quad + (I_1 + I_2 + \cdots + I_n) + J_m \\
&= \sum_{j=1}^{m} \sum_{i=1}^{n} \left(I_i + J_j\right)
\end{aligned}
$$

Theorem 3.12.1 above shows interval list disjunction can be broken down into two steps. We first find the disjunction of list L_1 with each interval J_j in turn comprising list L_2. We then find the disjunction of these intermediate results. This is made explicit by Algorithm 3.12.2 in Section 3.12.1 below.

3.12.1 Disjunction of Interval List With Interval

Inspection of Theorem 3.12.1 above hints at how we should proceed to implement interval list disjunction. We first define how to find the disjunction of an interval list with a single interval. We require an auxiliary function "*concat*" which appends a single interval J to the end of an interval list $L = \{I_1, I_2, \cdots, I_n\}$. Since interval lists are constructed from *disjoint* intervals, function "*concat*" may only be called when the resulting list maintains this constraint. That is, interval J does not overlap or touch any interval comprising list L.

Algorithm 3.12.1. ***Concatenation of Interval to Interval List**: Interval J is joined to the end of interval list $L = \{I_1, I_2, \cdots, I_n\}$ as defined by the following pseudo-code:*

```
$concat(L, J)$ return interval_list is
    return $\{I_1, I_2, \cdots, I_n, J\}$;
```

We now give an algorithm to find the disjunction of an interval list with a single interval. This algorithm calls Algorithm 3.8.1 (page 71), the interval disjunction function and function "*concat*" above.

Algorithm 3.12.2. ***Disjunction of Interval and Interval List**: Consider a single interval J and an interval list $L = \{I_1, I_2, \cdots, J_n\}$. The disjunction $dis(L, J)$ is given by the following pseudo-code:*

```
$dis(L, J)$ return interval_list is
    $J_{tmp} \leftarrow J$;
    $L_{tmp} \leftarrow \{\}$;
    for i in 1..n loop
        if $disjoint\big(J_{tmp}, I_i\big)$ then
            $L_{tmp} \leftarrow concat\big(J_{tmp}, I_i\big)$
        else
            $J_{tmp} \leftarrow dis\big(J_{tmp}, I_i\big)$
        end if;
    end loop;
    $L_{tmp} \leftarrow concat\big(L_{tmp}, J_{tmp}\big)$;
    return $L_{tmp}$;
```

The key functionality contained in Algorithm 3.12.2 above is that each time the interval J overlaps or touches the current interval I_i from list L, this modifies J according to the disjunction rule of Theorem 3.7.1 (page 67). If J and I_i are disjoint, then I_i is simply concatenated to the interval list to be returned.

3.12.2 Disjunction of Interval List With Interval List

We now complete the full algorithm of the disjunction of two interval lists by utilising Algorithm 3.12.2 above.

Algorithm 3.12.3. ***Disjunction of Interval Lists**: Consider interval lists $L_1 = \{I_1, I_2, \cdots, I_n\}$ and $L_2 = \{J_1, J_2, \cdots, J_m\}$. The disjunction $dis(L_1, L_2)$ of interval list L_1 with interval list L_2 is given by the following pseudo-code:*

```
$dis(L_1, L_2)$ return interval_list is
```

```
$L_{tmp} \leftarrow L_1;$
$for\ i\ in\ 1..m\ loop$
   $L_{tmp} \leftarrow dis\big(L_{tmp}, J_i\big);$
$end\ loop;$
$return\ \ L_{tmp};$
```

We note that the binary operation disjunction with respect to interval lists is *closed*, since the result of the disjunction operation is always another interval list.

3.12.3 Complexity of Interval List Disjunction

We now describe the complexity of our interval list disjunction algorithm. We proceed in three steps, beginning with the basic interval disjunction algorithm (Algorithm 3.8.1, page 71), then to disjunction with an interval list and an interval (Algorithm 3.12.2) and finally to disjunction of two interval lists (Algorithm 3.12.3).

- The basic interval disjunction algorithm of Theorem 3.7.1 consists in essence of two comparison operations ("lower of left bounds, higher of right bounds") in the worst case. We will use constant *c* to designate this complexity.

- Now suppose we have an interval list L_1 made up of intervals $I_1, I_2, \cdots, I_n$ and an interval J. Then the complexity of the disjunction algorithm $dis(L_1, J)$ of Algorithm 3.12.2 is given by *nc*, since we repeat the basic operations *n* times.

- Now we replace interval J with another interval list $L_2 = \{J_1, J_2, \cdots, J_m\}$. The disjunction algorithm $dis(L_1, L_2)$ of Algorithm 3.12.3 therefore has complexity given by *mnc* since we repeat the operations of the Algorithm 3.12.2 algorithm *m* times.

From the above we conclude the worst case complexity of interval list disjunction is *O*(*mn*) where *m* and *n* are the respective number of intervals in each interval list. Therefore interval list disjunction has polynomial complexity.

The design of Algorithm 3.12.3 allows for a parallel implementation. If the call to Algorithm 3.12.2 is carried out in parallel, then the complexity of interval list disjunction approaches *O*(*kn*) where *n* is the number of intervals in the first interval list and *k* is some constant denoting the cost of a single call to Algorithm 3.12.2.

3.13 Conjunction of Interval Lists

In this section we extend the definition of interval conjunction from Sections 3.6 (page 63) to apply to interval *lists*. We make use of the summation notation from Section 3.8.1 (page 70). The conjunction of interval lists on the Real number line is depicted in Figure 3.13 as the intersection of the two lists.

Theorem 3.13.1. *Consider interval lists* $L_1 = \{I_1, I_2, \cdots, I_n\}$ *and* $L_2 = \{J_1, J_2, \cdots, J_m\}$. *Then the conjunction of these lists* $L_1 \cdot L_2$ *is given by:*

$$L_1 \cdot L_2 = \sum_{j=1}^{m} \sum_{i=1}^{n} \left(I_i \cdot J_j \right)$$

Proof:

$$\begin{aligned}
L_1 \cdot L_2 &= \sum_{i=1}^{n} I_i \;\cdot\; \sum_{j=1}^{m} J_j \\
&= (I_1 + I_2 + \cdots + I_n) \cdot (J_1 + J_2 + \cdots + J_m) \\
&= \quad (I_1 + I_2 + \cdots + I_n) \cdot J_1 \\
&\quad + (I_1 + I_2 + \cdots + I_n) \cdot J_2 \\
&\quad\; \vdots \\
&\quad + (I_1 + I_2 + \cdots + I_n) \cdot J_m \\
&= \sum_{j=1}^{m} \sum_{i=1}^{n} \left(I_i \cdot J_j \right)
\end{aligned}$$

Theorem 3.13.1 above shows interval list conjunction can be broken down into two steps. We first find the conjunction of list L_1 with each interval J_j in turn comprising list L_2. We then find the disjunction of these intermediate results. This is made explicit by Algorithm 3.13.1 in Section 3.13.1 below.

3.13.1 Conjunction of Interval List With Interval

Inspection of Theorem 3.13.1 above hints at how we should proceed to implement interval list conjunction. We first define how to find the conjunction of an interval list L with a single interval J. Intuitively, we apply the conjunction rule from Theorem 3.6.1 (page 64) using J and each interval comprising L in turn. Indeed, this follows immediately from the distributive property of Boolean algebra (Pohl & Shaw 1986). We also make use of the "*concat*" function from Algorithm 3.12.1 (page 81).

Algorithm 3.13.1. ***Conjunction of Interval and Interval List****: Consider a single interval* J *and an interval list* $L = \{I_1, I_2, \cdots, I_n\}$. *The conjunction* $con(L, J)$ *is given by the following pseudo-code:*

```
$con(L, J)$ return interval_list is
    $J_{tmp} \leftarrow J$;
    $L_{tmp} \leftarrow \{\}$;
    for i in $1..n$ loop
        $J_{tmp} \leftarrow con(J, I_i)$;
        $L_{tmp} \leftarrow concat\big(L_{tmp}, J_{tmp}\big)$ ;
    $end\ loop$;
    $return\ \ L_{tmp}$;
```

We may use the "*concat*" function with confidence in Algorithm 3.13.1 above because the interval J_{tmp} which results from the conjunction of J and I_i can never expand I_i. Hence, J_{tmp} will never overlap any interval in L_{tmp}.

3.13.2 Conjunction of Interval List With Interval List

We now complete the full definition of the conjunction of two interval lists by utilising Algorithm 3.13.1 above. This definition also necessarily calls the disjunction function of Algorithm 3.12.3 above, which is why it was defined first. This follows from the observation of Section 3.8 (page 69) that an interval list is a disjunct of intervals.

Algorithm 3.13.2. ***Conjunction of Interval Lists**: Consider interval lists $L_1 = \{I_1, I_2, \cdots, I_n\}$ and $L_2 = \{J_1, J_2, \cdots, J_m\}$. The conjunction $con(L_1, L_2)$ is given by the following pseudo-code:*

```
$con(L_1, L_2)$ $return\ interval\_list\ is$
    $L1_{tmp} \leftarrow \{\}$;
    $L2_{tmp} \leftarrow \{\}$;
    $for\ i\ in\ 1..m\ loop$
        $L1_{tmp} \leftarrow con(L_1, J_i)$ ;
        $L2_{tmp} \leftarrow dis\big(L2_{tmp}, L1_{tmp}\big)$ ;
    $end\ loop$;
    $return\ \ L2_{tmp}$;
```

We note that the binary operation conjunction with respect to interval lists is *closed*, since the result of the conjunction operation is always another interval list.

3.13.3 Complexity of Interval List Conjunction

We now describe the complexity of our interval list conjunction algorithm. We proceed in three steps, beginning with the basic interval conjunction algorithm (Algorithm 3.6.1, page 65), then to conjunction with an interval list and an interval (Algorithm 3.13.1) and finally to conjunction of two interval lists (Algorithm 3.13.2).

- The basic interval conjunction algorithm of Theorem 3.6.1 consists in essence of two comparison operations ("higher of left bounds, lower of right bounds") in the worst case. We will use constant c to designate this complexity.

- Now suppose we have an interval list L_1 made up of intervals $I_1, I_2, \cdots, I_n$ and an interval J. Then the complexity of the conjunction algorithm $con(L_1, J)$ of Algorithm 3.13.1 is given by nc, since we repeat the basic operations n times.

- Now we replace interval J with another interval list $L_2 = \{J_1, J_2, \cdots, J_m\}$. The conjunction algorithm $con(L_1, L_2)$ of Algorithm 3.13.2 therefore has complexity given by mnc since we repeat the operations of the Algorithm 3.13.1 algorithm m times. However, the conjunction algorithm itself calls the disjunction algorithm m times. Therefore, using the complexity result we derived in Section 3.12.3 for interval list disjunction, we must replace the "m" with "mn", yielding a complexity of $(mn)\,nc = n^2mc$.

From the above we conclude the worst case complexity of interval list conjunction is $O\left(n^2m\right)$ where m and n are the respective number of intervals in each interval list. Therefore interval list conjunction has polynomial complexity.

The design of Algorithm 3.13.2 allows for a parallel implementation. If the call to Algorithm 3.13.1 is carried out in parallel, then the complexity of interval list conjunction approaches $O\left(kn^2\right)$ where n is the number of intervals in the first interval list and k is some constant denoting the cost of a single call to Algorithm 3.13.1.

3.14 Negation of Interval Lists

In this section we extend the definition of interval negation from Section 3.9 (page 72) to apply to interval *lists*. We make use of the summation notation and product notation from Section 3.8.1 (page 70). The negation of an interval list on the Real number line is depicted in Figure 3.14 as the complement of the list.

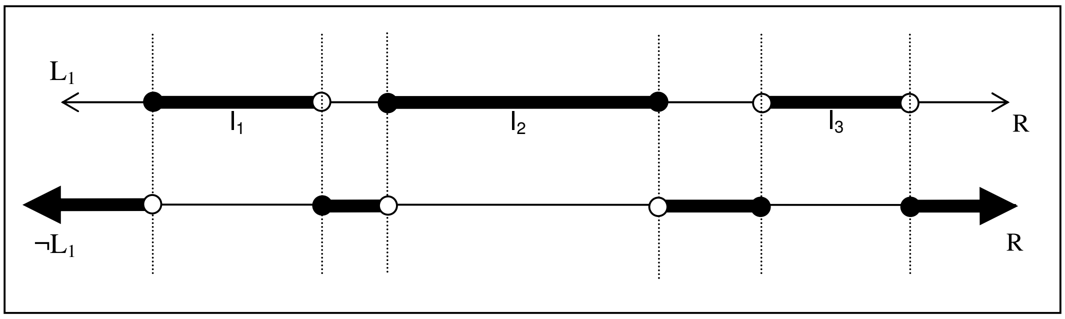


Figure 3.14: **Interval list negation**: We may represent the negation of an interval list L_1 as the *complement* of the list.

Theorem 3.14.1. *Consider and interval lists $L = \{I_1, I_2, \cdots, I_n\}$. Then the negation of this list $\neg L$ is given by:*

$$\neg L = \prod_{i=1}^{n} \neg I_i$$

***Proof*:**

$$\begin{aligned}
\neg L &= \neg \left(\sum_{i=1}^{n} I_i \right) \\
&= \neg \left(I_1 + I_2 + \cdots + I_n \right) \\
&= \neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \\
&= \prod_{i=1}^{n} \neg I_i
\end{aligned}$$

We arrive at the last line by applying De Morgan's law (Pohl & Shaw 1986) and find the conjunct of the negated intervals making up the list L. The above theorem leads directly to the following algorithm which calls the interval negation function of Algorithm 3.9.1 (page 74) and the interval list conjunction function of Algorithm 3.13.2.

Algorithm 3.14.1. ***Negation of Interval List*:** *neg* (L)*: Consider an interval list* L *comprised of disjoint intervals* $I_1, I_2, \cdots, I_n$*. Then the negation neg* (L) *is given by the following pseudo-code:*

```
$neg\,(L)$ $return\ interval\_list\ is$
    $L1_{tmp} \leftarrow \{\}$;
    $L2_{tmp} \leftarrow \{\mathbf{1}\}$;
    $for\ i\ in\ 1..n\ loop$
        $L1_{tmp} \leftarrow neg\,(I_i)$;
        $L2_{tmp} \leftarrow con\left(L1_{tmp}, L2_{tmp}\right)$;
    $end\ loop$;
    $return\ L2_{tmp}$;
```

We note that the unary operation negation with respect to interval lists is *closed*, since the result of the negation operation is always another interval list.

3.14.1 Complexity of Interval List Negation

We now describe the complexity of our interval list negation algorithm. We proceed in two steps, beginning with the basic interval negation algorithm (Algorithm 3.9.1, page 74), then to negation with an interval list (Algorithm 3.14.1).

- The basic interval negation algorithm of Algorithm 3.9.1 consists in essence of two table look ups to find the negation of the left and right limits respectively. We will use constant c to designate this complexity.
- Now suppose we have an interval list L made up of intervals $I_1, I_2, \cdots, I_n$. Then the worst case complexity of the negation algorithm *neg*(L) of Algorithm 3.14.1 is given by nc, since we repeat the basic operations n times.

However, the negation algorithm itself calls the conjunction algorithm (which in turn calls the disjunction algorithm) n times. Therefore, using the complexity result we derived in Section 3.13.3, we must replace the "n" term with "n^2m", yielding a complexity of n^3c.

From the above we conclude the worst case complexity of interval list negation is $O\left(n^3\right)$ where n is the number of intervals in the interval list. Therefore interval list negation has polynomial complexity. The design of Algorithm 3.14.1 does not allow for a parallel implementation because it progressively accumulates the result of the conjunction of the negated intervals.

3.15 Special Interval Lists

We now define two special interval lists: the *infinite interval list* and the *null interval list*, which are analogous to the *infinite interval* of Section 3.5.1.1 (page 60) and the *null interval* of Section 3.5.1.2 (page 61). We then show how these special interval lists act as identity elements for interval lists, in an analogous fashion to Theorems 3.10.1 and 3.10.2 (page 74).

3.15.1 The Infinite Interval List

Informally, we may picture the infinite interval list in an analogous manner to the infinite interval of Section 3.5.1.1 (page 60). That is, it is a list of intervals that encloses all values in the domain. We use the infinite interval symbol **1** to represent the infinite interval list as well, relying on context to make the distinction. Ultimately both of these notations are equivalent to the Boolean constant "`true`". A precise definition follows.

Definition 3.15.1. ***The infinite interval list****: This is the interval list consisting of a single infinite interval. That is:*

$$\begin{aligned}\mathbf{1} &= \sum_{i=1}^{n} \mathbf{1}\\ &= \texttt{true}_1 + \texttt{true}_2 + \cdots + \texttt{true}_n && \textit{by Thm 3.5.1}\\ &= \{\mathbf{1}\}\end{aligned}$$

According to the definition above, the infinite interval list comprises a list of *at least one* infinite interval. Since we insist the intervals comprising an interval list are disjoint, we always reduce such a list to a single infinite interval. We now show the infinite interval list acts as the *identity element* for the conjunction of interval lists.

Theorem 3.15.1. *The infinite interval list* **1** *is the identity element when we apply the binary operator "con" (conjunction) to interval lists.*
Proof: *Let $L = \{I_1, I_2, \cdots, I_n\}$ be an arbitrary interval list comprising n disjoint intervals. Then by Theorem 3.13.1 the conjunction of this list with the infinite interval list* **1** *is given by:*

$$\begin{aligned} L \cdot \mathbf{1} &= \sum_{j=1}^{m} \sum_{i=1}^{n} (I_i \cdot \mathbf{1}) \\ &= \sum_{i=1}^{n} (I_i) \qquad \textit{by Thm 3.10.1} \\ &= L \end{aligned}$$

Similarly

$$\begin{aligned} \mathbf{1} \cdot L &= \sum_{j=1}^{n} \sum_{i=1}^{m} \left(\mathbf{1} \cdot I_j\right) \\ &= \sum_{j=1}^{n} \left(I_j\right) \qquad \textit{by Thm 3.10.1} \\ &= L \end{aligned}$$

Since L was an arbitrary interval, **1** *is the identity element for interval lists under the operation "con" (conjunction).*

3.15.2 The Null Interval List

Informally, we may picture the null interval list in an analogous manner to the null interval of Section 3.5.1.2 (page 61). That is, it is a list of intervals that never encloses any values. We use the null interval symbol **0** to represent the null interval list as well, relying on context to make the distinction. Ultimately both of these notations are equivalent to the Boolean constant "`false`". A precise definition follows.

Definition 3.15.2. ***The null interval list* 0**: *This is the interval list consisting of* $0..n$ *null intervals. That is:*

$$\mathbf{0} = \sum_{i=0}^{n} \mathbf{0}$$

According to the definition above, the null interval list consists of *zero* or more null intervals. Therefore, the empty interval list containing no intervals is also by definition equivalent to the null interval list. We now show the null interval list acts as the identity element for the disjunction of interval lists.

Theorem 3.15.2. *The null interval list* $\{\mathbf{0}\}$ *is the identity element when we apply the* *binary operator "dis" (disjunction) to interval lists.*

***Proof**: Let $L = \{I_1, I_2, \cdots, I_n\}$ be an arbitrary interval list comprising n disjoint intervals. Then by Theorem 3.12.1 the disjunction of this list with the null interval list* **0** *is given by:*

$$\begin{aligned} L + \mathbf{0} &= \sum_{j=0}^{m}\sum_{i=1}^{n}(I_i + \mathbf{0}) \\ &= \sum_{i=1}^{n}(I_i) \qquad \textit{by Thm 3.10.2} \\ &= L \end{aligned}$$

Similarly

$$\begin{aligned} \mathbf{0} + L &= \sum_{j=1}^{n}\sum_{i=0}^{m}\left(\mathbf{0} + I_j\right) \\ &= \sum_{j=1}^{n}\left(I_j\right) \qquad \textit{by Thm 3.10.2} \\ &= L \end{aligned}$$

Since L was an arbitrary interval, **0** *is the identity element for interval lists under the operation "dis" (disjunction).*

3.16 Interval List Subsumption and Implication

In this section we define two further Boolean functions, *sub* (subsumes) and *imp* (implies) which operate on our interval list data type. This is an extension of Section 3.11 (page 77) where we defined subsumption and implication for intervals.

3.16.1 Interval List Subsumption

We now consider subsumption of interval lists. Consider Figure 3.15. We firstly define precisely what we mean by subsumption of interval lists.

Definition 3.16.1. ***Subsumption of Interval Lists**: Consider two arbitrary interval lists L_1 and L_2. Interval list $L_1 = \{I_1, I_2, \cdots, I_n\}$ while interval list $L_2 = \{J_1, J_2, \cdots, J_m\}$. Then L_1* subsumes *L_2 if and only if every interval J_j making up list L_2 is subsumed by some interval I_i in L_1. That is:*

$$\forall J_j\ \exists I_i\ :\quad sub\left(I_i, J_j\right),\quad i = 1..n,\ j = 1..m$$

Using the above definition, we now derive a theorem for the subsumption of an interval J by an interval list L. In the following we use "*sub*" to denote "subsumes".

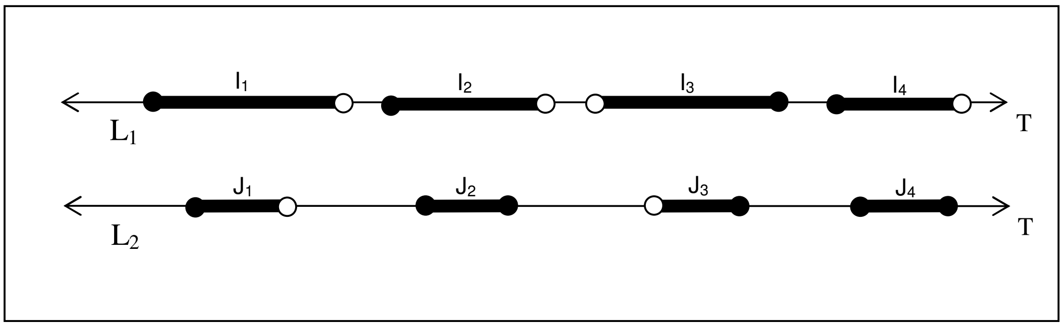


Figure 3.15: **Subsumption of interval lists**: Interval List L_2 is by definition *subsumed* by list L_1 if for every interval J_i making up list L_2 is subsumed by some interval in L_1.

Theorem 3.16.1. ***Subsumption of Interval by Interval List**: Consider an interval list $L = \{I_1, I_2, \cdots, I_n\}$ and an interval J. Then:*

$$L \; sub \; J \quad \Leftrightarrow \quad \sum_{i=1}^{n} (I_i \; sub \; J)$$

***Proof**: By Definition 3.16.1, L subsumes J if and only if J is subsumed by some interval I_i, $i = 1..n$. That is:*

$$\begin{aligned} L \; sub \; J \quad &\Leftrightarrow \quad \left(\sum_{i=1}^{n} I_i \right) sub \; J \\ &\Leftrightarrow \quad I_1 subJ \; + \; I_2 subJ \; + \; \cdots \; + \; I_n subJ \\ &\Leftrightarrow \quad \sum_{i=1}^{n} (I_i \; sub \; J) \end{aligned}$$

We now derive a theorem for the subsumption of an interval list L_2 by an interval list L_1.

Theorem 3.16.2. ***Subsumption of Interval List by Interval List**: Consider an interval list $L_1 = \{I_1, I_2, \cdots, I_n\}$ and an interval list $L_2 = \{J_1, J_2, \cdots, J_m\}$. Then:*

$$L_1 \; sub \; L_2 \quad \Leftrightarrow \quad \prod_{j=1}^{m} \sum_{i=1}^{n} \left(I_i \; sub \; J_j \right)$$

***Proof**: Asserting that "L_1 subsumes L_2" means precisely that L_1 subsumes J_j for* *all $j = 1..m$. That is:*

$$\begin{aligned}
L_1\ sub\ L_2 &\Leftrightarrow (L_1\ sub\ J_1)\cdot(L_1\ sub\ J_2)\cdot\ \cdots\ \cdot(L_1\ sub\ J_m)\\
&\Leftrightarrow (I_1 subJ_1\ +\ I_2 subJ_1\ +\ \cdots\ +\ I_n subJ_1)\\
&\quad\cdot(I_1 subJ_2\ +\ I_2 subJ_2\ +\ \cdots\ +\ I_n subJ_2)\\
&\quad\ \vdots\\
&\quad\cdot(I_1 subJ_m\ +\ I_2 subJ_m\ +\ \cdots\ +\ I_n subJ_m)\\
&\Leftrightarrow \prod_{j=1}^{m}\sum_{i=1}^{n}\left(I_i\ sub\ J_j\right)
\end{aligned}$$

We now use Theorems 3.16.1 and 3.16.2 to give an algorithm for the Boolean function “*sub*” (subsumes). We proceed in two steps, firstly defining a pseudo-code function for subsumption of interval J by interval list $L = \{I_1, I_2, \cdots, I_n\}$. This definition utilises the interval “*sub*” function defined in Algorithm 3.11.1 (page 78). We then define the full subsumption algorithm for interval lists.

Algorithm 3.16.1. ***Subsumption of Interval by Interval List****: $sub(L, J)$: Consider an interval list L comprised of disjoint intervals $I_1, I_2, \cdots, I_n$ and an interval J. Then the Boolean function “sub” (subsumes) is given by the following pseudo-code:*

```
sub(L, J) return boolean is
   tmp ← false;
   for i in 1..n loop
      tmp ← sub(I_i, J);
      if tmp then i ← n;
   end loop;
   return tmp;
```

We now use the above function to define the full subsumption algorithm for interval lists:

Algorithm 3.16.2. ***Subsumption of Interval List by Interval List****: $sub(L_1, L_2)$: Consider an interval list L_1 comprised of disjoint intervals $I_1, I_2, \cdots, I_n$ and an interval list L_2 comprised of disjoint intervals $J_1, J_2, \cdots, J_m$. Then the Boolean function “sub” (subsumes) is given by the following pseudo-code:*

```
sub(L_1, L_2) return boolean is
   tmp ← false;
   for j in 1..m loop
      tmp ← sub(L_1, J_j);
      if not tmp then j ← m;
   end loop;
   return tmp;
```

3.16.2 Interval List Implication

We now consider implication with interval lists. Consider again Figure 3.15. Our objective here is to capture the intuitive notion that if we assume the truth of the more restrictive interval list L_2 then we can logically assume the truth of the less restrictive interval list L_1 that subsumes L_2.

Theorem 3.16.3. ***Interval List Implication**: Consider two arbitrary interval lists $L_1 = \{I_1, I_2, \cdots, I_n\}$ and $L_2 = \{J_1, J_2, \cdots, J_n\}$ such that L_1 subsumes L_2. Then $L_2 \rightarrow L_1$.*
***Proof** (by contradiction): Assume that L_1 subsumes L_2 but that L_2 does not imply L_1. That is:*

$$\begin{aligned}
\neg(L_2 \rightarrow L_1) &= \neg(\neg L_2 + L_1) \\
&= L_2 \cdot \neg L_1 \\
&= \left(\sum_{j=1}^{m} J_j\right) \cdot \neg\left(\sum_{i=1}^{n} I_i\right) \\
&= \left(\sum_{j=1}^{m} J_j\right) \cdot \left(\prod_{i=1}^{n} \neg I_i\right) \\
&= \left(\prod_{i=1}^{n} \neg I_i\right) \cdot \left(\sum_{j=1}^{m} J_j\right) \\
&= (\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n) \cdot (J_1 + J_2 + \cdots + J_m) \\
&= \quad (\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \cdot J_1) \\
&\quad + (\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \cdot J_2) \\
&\quad\ \vdots \\
&\quad + (\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \cdot J_m)
\end{aligned}$$

Consider each of the terms in the disjunction above, for example the second term:

$$(\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \cdot J_2)$$

Interval J_2 must be subsumed by some I_i where $i = 1..n$. But this means $J_2 \rightarrow I_i$, by Theorem 3.11.2. So it cannot be the case that $\neg I_i \cdot J_2$. Therefore it must be that $(\neg I_1 \cdot \neg I_2 \cdot \cdots \cdot \neg I_n \cdot J_2)$ is `false`*. The same is true for each term in the disjunction. Therefore we have a contradiction. So it cannot be that L_2 does not imply L_1. Therefore L_2 does imply L_1.*

Algorithm 3.16.3. ***Implication with Interval Lists**: Consider two arbitrary interval lists $L_1 = \{I_1, I_2, \cdots, I_n\}$ and $L_2 = \{J_1, J_2, \cdots, J_m\}$. We now define the Boolean function "imp" (implication) of interval list L_1 by interval list L_2 with the following pseudo-code function.*

```
$imp(L_2, L_1)$ return boolean is
    return $sub(L_1, L_2)$;
```

3.17 Summary

The main objective of this chapter is to build the theoretical foundation for our Reasoning Engine. We achieve this by defining an *interval* data type around which we build an *interval algebra* which we show is analogous to Boolean Algebra. We use the *interval* data type to succinctly capture the notion of *the valid or legal range of values a variable may assume.* Starting with the simple assumption of the validity of Boolean Algebra, in addition to the basic assumption that the data types we utilise can be *totally ordered*, we show that it is straightforward to define all the object structures and behaviours we require.

The main contributions of this Chapter include the following.

- We begin this chapter by setting out our basic assumptions and working definitions. We take the Boolean Algebra as being axiomatic along with the ability to impose a deterministic total ordering on the data types we consider (Section 3.2).
- We then define the data structures *limit* (Section 3.3) and *bound* (Section 3.4) which we subsequently use to define our basic data structure, the *interval* (Section 3.5).
- We show that the interval is a contracted form of a sentence in first order logic consisting of the conjunction of two assertions: one concerning the left bound and the other concerning the right bound (Section 3.4.2).
- We then define three basic operations: *conjunction* (Section 3.6), *disjunction* (Section 3.7) and *negation* (Section 3.9) with intervals. We show that of these three operations, only conjunction is *closed* with respect to intervals.
- We give definitions for the special intervals *the infinite interval* and *the null interval* (Section 3.5.1) and show that these act as the identity elements **1** and **0** with respect to conjunction and disjunction of intervals (Section 3.10).
- We give algorithms for the Boolean binary operations *subsumption* and *implication* using intervals (Section 3.11).
- We define the *interval list* to be a disjunction of disjoint intervals (Section 3.8). We extend our definitions of conjunction (Section 3.13), disjunction (Section 3.12) and negation (Section 3.14) to apply to interval lists and show that these three operations are *closed* with respect to interval lists.

- We derive the complexity of the three interval list operations disjunction (Section 3.12.3), conjunction (Section 3.13.3) and negation (Section 3.14.1) and show them to have polynomial complexity in the worst case. In the case of disjunction and conjunction of interval lists, the algorithms allow for a parallel implementation which reduces the complexity.

- We give definitions for the special interval lists *the infinite interval list* and *the null interval list* and show that these act as the identity elements **1** and **0** with respect to conjunction and disjunction of interval lists (Section 3.15).

- We give algorithms for the Boolean binary operations *subsumption* and *implication* using interval lists (Section 3.16).

The operations we have defined in this chapter form the foundation of our Reasoning Engine, upon which we have built a practical semantic query optimizer. Our semantic optimizer is described in detail in Chapter 4.

Chapter 4

A Practical Semantic Query Optimizer

4.1 Introduction

In this chapter we describe the design of a practical semantic query optimizer. Our design anticipates some of the important conclusions we draw from our empirical study in that we propose to utilise those techniques from our research into semantic query optimization which are likely to have the most positive influence on the performance of query evaluation.

We begin by highlighting what we consider to be an intrinsic limiting factor of semantic query optimization *per se*. Much of SQO, by its very nature, is limited in its effectiveness by the fact that it depends on the detection of queries which are, in some sense, anomalous. But if anomalous queries are hardly ever submitted, perhaps the extra effort of semantically optimizing queries is not worthwhile. To our knowledge, this is the first study to specifically highlight this property of SQO.

Our semantic optimizer is designed to preprocess SQL queries and sits in front of the normal SQL parser and optimizer. We reiterate that much of the meta-data required for a simple but effective semantic query optimizer is already available within commercial RDBMS. We propose, for example, to make use of schema constraints that are encoded within the database but which are currently ignored during the process of query evaluation.

While many researchers advocate the mechanical discovery of semantic rules which might be of benefit to a semantic query optimizer, we argue that such a rule discovery exercise is unlikely to be optimal without a knowledge of the *query profile*[1] where we first discover what database objects are actually being queried. This knowledge can then be used to focus a subsequent rule discovery exercise.

Our semantic optimizer can utilise two types of semantic rule. The first type of semantic rule is *always true* and includes the schema constraints currently stored and maintained by the DBMS. Therefore, they may be added at any time to SQL query restrictions without altering the logical outcome of the query. We use this fact to rewrite the query such that it may be evaluated more efficiently. When we say a rule is “always true” we mean the rule is *true for the lifetime of the schema*. Therefore, we make the assumption that schema evolution is rare or does not occur at all. Such an assumption is quite reasonable, especially for data warehouses. We do not specifically address the issue of schema evolution with our practical semantic optimizer[2].

The second type of semantic rule is *sometimes true*. This includes the discovery of data “holes”[3] which we deduce will return zero rows[4]. In addition, we show

[1]See Definition 4.3.1, page 102.

[2]However, we deal with this topic in Chapter 2.

[3]See Definition 4.3.3, page 104.

[4]Hence the term *zero* queries. See Definition 4.3.2, page 104.

how *conditional rules* may be elegantly encoded using the interval list form we described in Chapter 3. This allows us, for example, to capture the knowledge of domain experts and use it to enhance the performance of the semantic optimizer. To our knowledge, this is the first study to report the use of intervals or interval lists in this manner.

When we say a rule is “sometimes true” we mean the validity of the rule depends on the database state remaining static; i.e., stored data is not modified for the duration the rule is utilised. This is because this type of rule is derived from an analysis of *data*, and is not a consequence of the application semantics.

Example 4.1.1. *A data analysis discovers a precise correlation between the values stored in two columns c_i and c_j of a table T which is part of a data warehouse. A semantic rule is formulated based on this correlation. The following evening, data in table T is refreshed, invalidating the rule derived the previous evening.*

The point of the above example is that any time data is updated, we run the risk of “sometimes true” rules being invalidated. Therefore, the cost of revalidating such rules must be taken into account in any usable semantic optimizer[5]. The data analysis we report in the search for such rules is simple and unlikely to require significant computational resources. Furthermore it is carried out “off-line” and so does not impact on query evaluation.

The remainder of this Chapter is organised as follows.

- We begin by highlighting an intrinsic limiting factor of semantic query optimization (Section 4.2).
- We then introduce four new terms: *query profile*, *zero query*, *positive query* and *data holes* (Section 4.3).
- We follow with a description of how our practical semantic query optimizer functions as a preprocessor, sitting in front of the normal SQL query optimizer (Section 4.4).
- We describe how we define meta-data for use with our semantic optimizer (Section 4.4.1).
- We propose a new type of semantic query optimization which searches for “data holes” and utilises them to identify zero queries which, in an analogous fashion to unsatisfiable queries, need not be submitted to the database (Section 4.4.2).

[5]This problem was described in Section 2.3, page 19.

- We describe in detail how we harvest a subset of existing schema constraints which are already stored as part of the RDBMS and how these are utilised by our semantic optimizer (Section 4.4.3).
- We then explain how the optimizer may be extended with conditional rules which are derived from a data driven analysis and which typically capture correlations between non-indexed and indexed columns (Section 4.5). These rules may be elegantly expressed as interval lists and are invoked by application of the Subsumption Rule (Section 4.5.2).
- Finally, we summarise the main contributions of the Chapter (Section 4.6).

4.2 An Intrinsic Limitation of SQO

In this section we focus on an intrinsic limiting factor of semantic query optimization *per se*. That is, much of semantic query optimization, by its very nature, is limited in its effectiveness by the fact that it depends on the detection of queries which are, in some sense, anomalous. The detection of unsatisfiable queries, sometimes described as the "ultimate win" for SQO (Godfrey et al. 2001), is a clear example. It is easy to see how preventing unsatisfiable queries from being submitted to the database might result in more efficient query processing. However, if unsatisfiable queries are never (or hardly ever) made against the target database schema, then perhaps the extra processing required to detect unsatisfiable queries is not worthwhile[6].

A similar argument can be made regarding the detection of out of range queries or queries which are satisfiable but nevertheless return zero rows because they target "holes"[7] in the data. If such queries are hardly ever submitted then perhaps the effort expended in detecting such queries is ultimately of little or no value.

We explicitly acknowledge this limitation on the usefulness of SQO and we regard it as an intrinsic property of SQO itself. To our knowledge, this is the first study to specifically highlight this property of SQO.

However, we do not think this built in limitation negates the potential of SQO. The remainder of this Section explains why this is so.

4.2.1 Utility of SQO

We now set out why SQO may be useful even in environments where anomalous queries are hardly ever submitted. We look firstly at the potential cost to the

[6]There are, to our knowledge, no published studies which research the relative prevalence of anomalous queries in industry RDBMS.

[7]See Definition 4.3.3, page 104.

database environment of submitting unsatisfiable queries. Secondly, we describe the impact on data integrity of relaxing schema constraints during bulk insert of new data. Thirdly, we briefly examine the consequences of automatically generated SQL queries. Finally, we consider the likelihood of sub-optimal SQL queries against view definitions in typical data warehouse schemas.

4.2.1.1 Unsatisfiable queries may be costly

Consider the case where the probability of an unsatisfiable join being submitted against a pair of large target tables is historically very low (say, less than 1%). However, this probability says nothing about the impact that such a query might have. If such a query is submitted and the target tables are large enough, this has the potential to take over a large proportion of available computational resources, negatively impacting on other database users and processes. It is easy to imagine that preventing such a situation from ever happening might itself make worthwhile the effort of semantically optimizing all queries.

4.2.1.2 Impact of relaxing schema constraints

Consider the case where schema constraints encoded within the database are relaxed whenever tables are populated with new data. This commonly occurs in data warehouses where tables are large and regularly refreshed in "batch mode" with large volumes of data. If schema constraints are enforced in such circumstances, the constraint is triggered and checked for each new row of data, resulting in greatly increased processing times. However, if the constraint is relaxed, there is a penalty to be paid for saving time: data integrity may never be checked.

Example 4.2.1. *In the Oracle RDBMS, data may be validated against a schema constraint* after *the new data has been inserted. This guarantees data integrity but may still require more time to complete than is acceptable. There is a further, potentially more serious problem, which is that if* any *data row fails the integrity check, the entire batch of new data is rolled back (Ashdown 2005*a*). Thus it may be impractical to enforce schema constraints during bulk insert of new data.*

We emphasize that no commercial RDBMS currently enforces schema constraints at query time. Therefore, if schema constraints have been relaxed during data insert or update, data integrity cannot be guaranteed unless constraints have been re-enabled and integrity checks allowed to proceed. It is easy to imagine how this uncertain, ambiguous situation might give rise to erroneous query results.

Example 4.2.2. *Consider a data warehouse which performs nightly bulk loads of sales figures from a number of regional stores, consolidating the data into a single* `SALES` *table. Data volumes are large so various schema constraints are relaxed* *during the bulk load of new data to ensure the operation completes before the beginning of the next business day. One of the constraints which is temporarily disabled in this way checks that all prices charged for items sold are positive and less than a sensible maximum value:*

```
check (UNIT_PRICE > 0 and UNIT_PRICE <= 5000);
```

However, at one store invoices have been cancelled by overwriting the price charged with −999*. While this practice is allowed for by the local software, it is not detected by the data warehouse software. Thus all total sales calculations performed by the data warehouse are likely to contain considerable error.*

A semantic query optimizer that harvests schema constraints resolves the problem of relaxed schema constraints. That is, even in the presence of data that violates schema integrity, semantically optimizing the queries guarantees that all query answers actually conform to the schema semantics. This is because schema constraints are effectively enforced at query time by the semantic query optimizer.

4.2.1.3 Automatically generated queries

We now briefly consider the impact of automatically generated queries. SQL queries may be automatically generated by GUI[8] tools which provide a visual interface into the underlying database. These include query builders[9] and report writers[10]. The aim of such tools is often to provide non-specialists with a "point and click" methodology to construct database queries without the necessity of knowing SQL. Typically, the user points to objects in the database and then establishes a relationship between them (such as identifying a join column) before submitting the query and receiving the results in a "user friendly" format. It is easy to imagine that the result of automatically translating the query of a naïve user might result in a highly sub-optimal SQL query, such as a cartesian product on two large tables. A semantic query optimizer might be employed in such a situation as a *query conditioner*, filtering naïve queries and applying schema knowledge to rewrite queries into more sensible alternatives. For example, rather than joining the requested tables directly, a semantic optimizer might instead consult a materialized view (DeHaan, Larson & Zhou 2005, Zaharioudakis, Cochrane, Lapis, Pirahesh & Urata 2000).

4.2.1.4 Sub-optimal queries against views

In Example 2.5.7 (page 37), we showed how redundant joins may arise in queries against tables which are related via a primary key and corresponding foreign key

[8]GUI: "graphical user interface"

[9]See for example Oracle's "Query Builder"(`http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b16373/qry_bldr.htm`).

[10]See for example "Crystal Reports" (`http://www.crystalreports.co.uk/`)

column. Indeed this is one of the main targets for optimization, *join elimination*, identified by SQO researchers (see Section 2.5.4, page 36). However, sub-optimal queries may arise naturally in the case where the query is made against a view comprising one or more table joins, for example in data warehouses built with a *star schema*[11] (Cheng et al. 1999).

Example 4.2.3. *Consider Figure 4.4 (page 108), which depicts part of a data warehouse. Suppose we have created the view* `CUSTOMER_SUMMARY` *with the following DDL:*

```
create view CUSTOMER_SUMMARY(
  KEY,
  NAME,
  ORDER_COUNT,
  TOTAL_QUANTITY
) as
  select c.KEY, c.NAME, COUNT(1), SUM(s.QUANTITY)
  from CUSTOMER c, SALES s
  where s.CUSTOMER_KEY = c.KEY
  group by c.KEY, c.NAME;
```

Now we pose the following query against the view:

```
select *
from   CUSTOMER_SUMMARY
where  KEY = 2006;
```

The SQL optimizer must rewrite the above query to consult the base table `SALES`*. A naïve execution plan might produce the following:*

```
select   c.KEY, c.NAME, COUNT(1), SUM(s.QUANTITY)
from     CUSTOMER c, SALES s
where    s.CUSTOMER_KEY = c.KEY
group by s.CUSTOMER_KEY
having   s.CUSTOMER_KEY = 2006;
```

If this execution plan is followed, all aggregates based on `CUSTOMER_KEY` *will first be calculated, then all rows eliminated except for the single row which satisfies the* `HAVING` *restriction. This would be very inefficient, so let us assume the optimizer applies the rules of the relational algebra and pushes the query's restriction "*`where KEY = 2006`*" into the* `WHERE` *clause. The result would therefore be the following more efficient query:*

[11]See Example 4.4.1, page 106 for a detailed description of a star schema.

```
select   c.KEY, c.NAME, COUNT(1), SUM(s.QUANTITY)
from     CUSTOMER c, SALES s
where    s.CUSTOMER_KEY = c.KEY
and      c.KEY = 2006
group by s.CUSTOMER_KEY;
```

However, we can still do better. A semantic optimizer might deduce that since `c.KEY` is in fact the primary key of table `CUSTOMER`, the sort triggered by the `group by` clause is redundant.

The main point of this example is that there is nothing intrinsically wrong or inefficient about either the view definition or the query against the view. Nevertheless, a sub-optimal execution path was produced in the absence of knowledge of the schema semantics.

4.3 Additional Helpful Definitions

We now introduce four new definitions, *query profile*, *zero query*, *positive query* and *data holes*, which we will utilise in the remainder of this Chapter.

4.3.1 Query Profile

Anecdotal evidence suggests that in business applications that utilise RDBMS, it is often the case that most query activity is based around a small subset of the tables. For example, in data warehouses one large aggregated table may be the target of all queries.

Definition 4.3.1. ***Query profile (QP)***: *A query profile is a high level description of queries actually made against the target database. Such a profile would include as a minimum:*

- *the tables actually queried;*
- *the columns cited in query restriction clauses;*
- *the join columns in any table joins.*

We use this new term to refer to a query analysis whose aim is not to identify a particular result set, but rather to identify SQO strategies which are likely to enhance query efficiency. For example, at its simplest level, a QP notes which objects have actually been queried. This is valuable information; we now know what objects should be targeted for optimization. Finding a QP is distinctly different from the query driven rule discovery defined in Definition 2.3.4 (page 22). We do not attempt to formulate semantic rules, only to identify suitable starting points for the discovery of such rules.

Discovery of QP is analogous to the rule discovery phase[12] advocated by (Shekhar et al. 1993) where it is envisaged that semantic knowledge is discovered from the database and converted into semantic rules which may then be utilised by the semantic query optimizer. While other writers suggest knowledge discovery might be guided by what queries are actually made, we make the stronger claim that discovery of the query profile ought to be a pre-requisite for rule discovery and strongly influence its focus. This is because one may infer suitable starting points in the search for *relevant* (Definition 2.3.5, page 23) semantic information. This is equivalent to an initial heavy pruning of the space of possible rules, making it much more likely discovered rules are relevant. We note that the capture of a query profile is already a normal part of DBA activities and it is easy to capture a simple QP using available software.

Example 4.3.1. *In the Oracle RDBMS, several methods are available to access details of queries actually made against the database and their computational cost. We now briefly describe two of these.*

1. *The view `V$SQLSTATS` contains resource usage information for all SQL statements that have been recently executed. For example, querying column `SQL_TEXT` will yield the first 1000 characters of all recent SQL queries. Crucially, these statements can be ordered by cost. For example, one may order by column `BUFFER_GETS` to detect high CPU using statements, by `DISK_READS` to detect high disk I/O or by `SORTS` to detect queries requiring sorting (Chan 2006*b*).*

2. *The software tool `TKPROF` reports each SQL statement executed along with the resources it has consumed, the number of times it was called, and the number of rows which it processed. This information may be automatically accumulated in an operating system file over an arbitrary period of time and may include the resources utilised by one or many simultaneous sessions accessing the target database. `TKPROF` is described in more detail in Chapter 5 and is the main method by which computational cost is measured in our own empirical investigations (Chan 2006*c*).*

4.3.2 Zero Queries, Positive Queries and Data Holes

We noted in Definition 2.2.6 (page 18) that unsatisfiable queries are logically excluded (for example, by the schema semantics) from returning any rows. Detection

[12] See Definition 2.3.4, page 22.

of unsatisfiable queries is identified as pivotal by all researchers into SQO (Yoon et al. 1999, Genet & Dobbie 1998, Zhang & Ozsoyoglu 1997, Hsu & Knoblock 1996, Godfrey & Gryz 1996, Illarramendi et al. 1994) because such queries, if they can be detected, need not be submitted to the database at all. Thus the potential exists for considerable savings in query cost, provided the cost of detecting the unsatisfiability is small in comparison to the cost of retrieving the empty answer set from the database. In practice, as our empirical results in Chapter 6 confirm, the major efficiency gain in this situation is, unsurprisingly, the suppression of unnecessary disk activity (Siegel et al. 1992).

In addition to queries which are unsatisfiable because of the schema semantics, there may also be queries which return zero rows simply because there is no data currently residing in the database which satisfy the query restrictions.

Definition 4.3.2. ***Zero Query****: A zero query is one which is unsatisfiable because the query restrictions cannot be satisfied by data currently residing in the database in its current state.*

A zero query returns no rows, not because these are logically excluded by the schema semantics, but because the query restrictions cannot be satisfied by data currently residing in the database. It is helpful to think of these queries as targeting gaps or holes in the data. In (Rishe, Sun & Barton 2003), the authors describe a data mining algorithm which aims to discover empty rectangles in two dimensional data and suggest that this "empty space knowledge" might be exploited by a semantic query optimizer. We describe the exploitation of zero queries for exactly this purpose in Section 4.4.2 (page 109). This idea leads to the following definition.

Definition 4.3.3. ***Data Holes****: A data hole is an interval for which no data currently exists in the database.*

We use the term "interval" in the above definition exactly in the sense that we have defined the term in Definition 3.5.1 (page 59). Our purpose is to identify value ranges for which we can be sure a zero query will result. This is described in detail below in Section 4.4.2 (page 109).

For completeness, we also define the new term: *positive query*.

Definition 4.3.4. ***Positive Query****: A positive query is one which returns at least one row, for a given database state.*

4.4 Semantic Query Optimizer As Preprocessor

In this section we sketch the design of a practical semantic query optimizer. Our optimizer sits in front of the normal SQL parser and optimizer and preprocesses the queries based on semantic rules stored in the database as meta-data. Figure 4.1 (page 105) illustrates this design. The semantic query optimizer itself is comprised

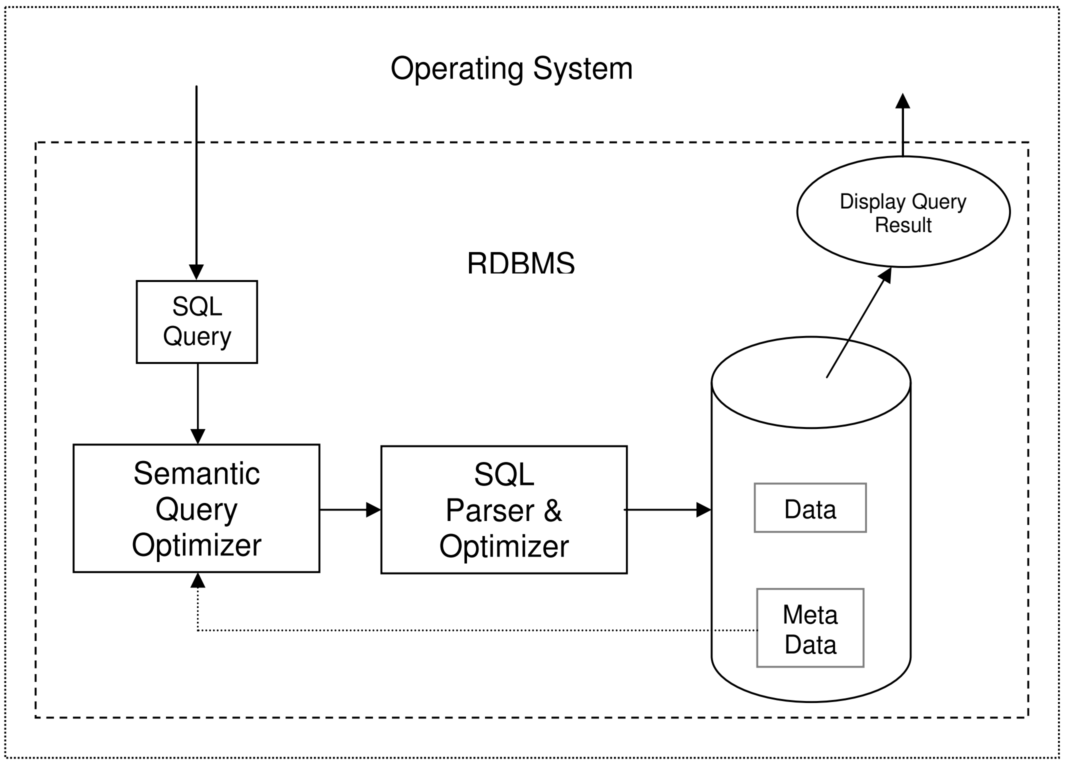


Figure 4.1: **Semantic query optimizer as preprocessor**: The semantic optimizer sits in front of the normal SQL parser and optimizer and preprocesses the queries based on semantic rules stored in the database as meta-data.

of the Reasoning Engine (RE) at its base combined with software layers for the collection and definition of meta-data plus a simple query preprocessing interface. This is illustrated in Figure 4.2 (page 106). The functions performed by the meta-data and preprocessing layers are illustrated in the use case diagram of Figure 4.3 (page 107). Through this interface, users may define meta-data for the target table objects. In addition, various types of semantic query optimization may be switched in and out.

4.4.1 Defining Meta-data

In this section we describe the process of collecting the meta-data which eventually will be utilised by the RE to semantically preprocess SQL queries. We present this as a typical series of steps which we carry out as a preliminary to the invocation of the optimizer itself.

1. **Collect Query Profile**: We argue that the first step in any effective implementation of SQO should be an examination of the table objects which are

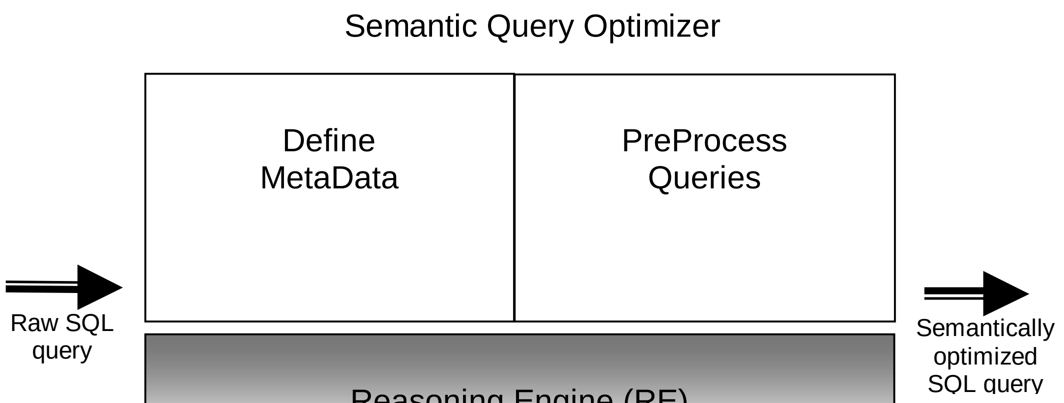


Figure 4.2: **Semantic query optimizer**: The semantic optimizer consists of the Reasoning Engine at its base plus software layers for the definition of meta-data and the preprocessing of queries.

actually being queried. Anecdotal evidence strongly suggests that it is frequently the case that a small subset of the tables making up a schema are actually the query targets. A basic query profiler might note the following information:

- which tables are actually queried;
- which table pairs are joined and the join columns;
- the restrictions applied to both queries and joins. In particular, the columns that appear in the restriction clauses are noted.

Example 4.4.1. *Consider Figure 4.4 (page 108), which depicts part of a data warehouse. The warehouse includes a summary table called* `SALES` *which summarises all sales made by customer, date, product code and invoice number. The configuration illustrated in Figure 4.4 is often described as a star schema. In the context of data warehouse design, table* `SALES` *is called the fact table while tables* `PRODUCT`*,* `CUSTOMER`*,* `SALES_DATE` *and* `INVOICE` *are called dimension tables. In this configuration, the primary key of table* `SALES` *is formed by concatenating the foreign keys pointing to each dimension table.*

A query profile discovers that a large number of queries are made against the `SALES` *table, with the foreign key columns (*`PRODUCT_KEY`*,* `CUSTOMER_KEY`*,* `DATE_KEY`*,* `INVOICE_NO`*) most often appearing in the query restrictions. The foreign key columns are therefore individually indexed and targeted for further analysis.*

In commercial RDBMS, this information is relatively easy to obtain without special software. For example, in the case of the Oracle RDBMS, this infor-

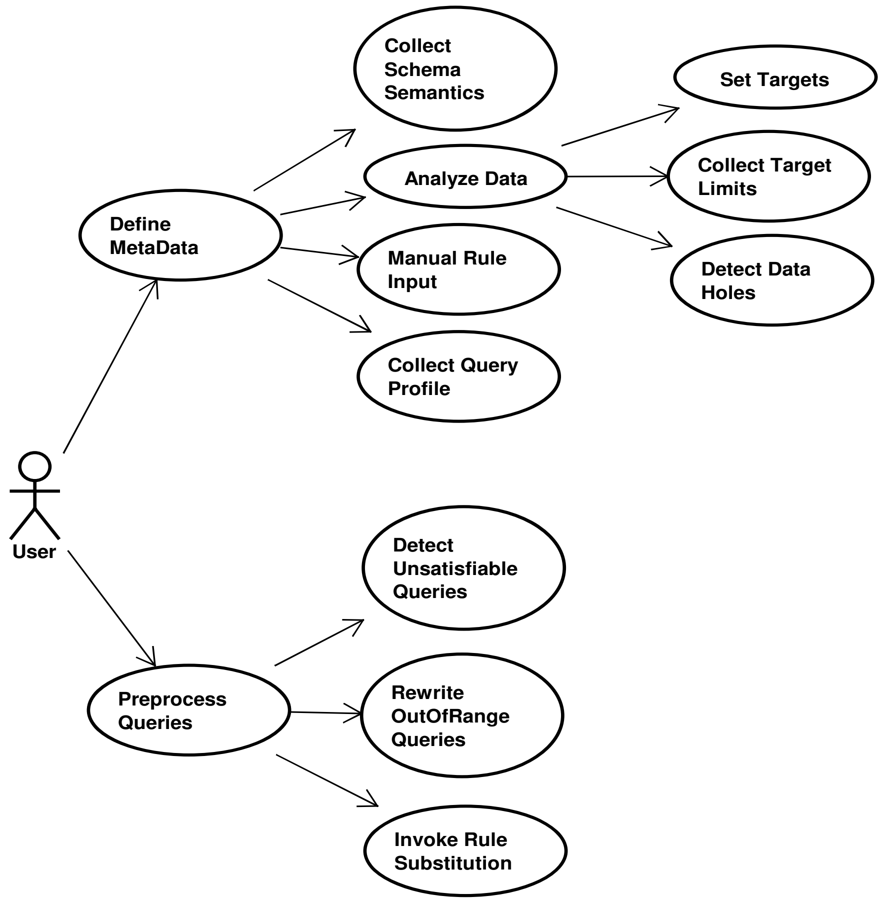


Figure 4.3: **Main functions of semantic query optimizer**: Through this interface, users may define meta-data for the target table objects. In addition, various types of semantic query optimization may be switched in and out.

mation can be collected with great accuracy over arbitrary periods of time and then analyzed with a standard software tool such as `TKPROF`[13].

2. **Harvest Schema Constraints**: Our next step is to harvest the various constraints defined and stored as part of the normal RDBMS meta-data. The default setting makes the assumption that if the various constraints already exist as part of the target schema, they are also worthwhile to harvest for the purposes of our semantic optimizer. However, the harvesting of schema constraints can also be restricted to a subset of tables identified by the query profile collected in Step 1 above. We describe the harvesting of schema constraints in more detail below in Section 4.4.3 (page 113).

3. **Analyze Data**: After collecting a query profile and harvesting the existing schema semantics, we then target the table objects identified by the query

[13]See Chapter 5 for a more complete description of Oracle's `TKPROF`.

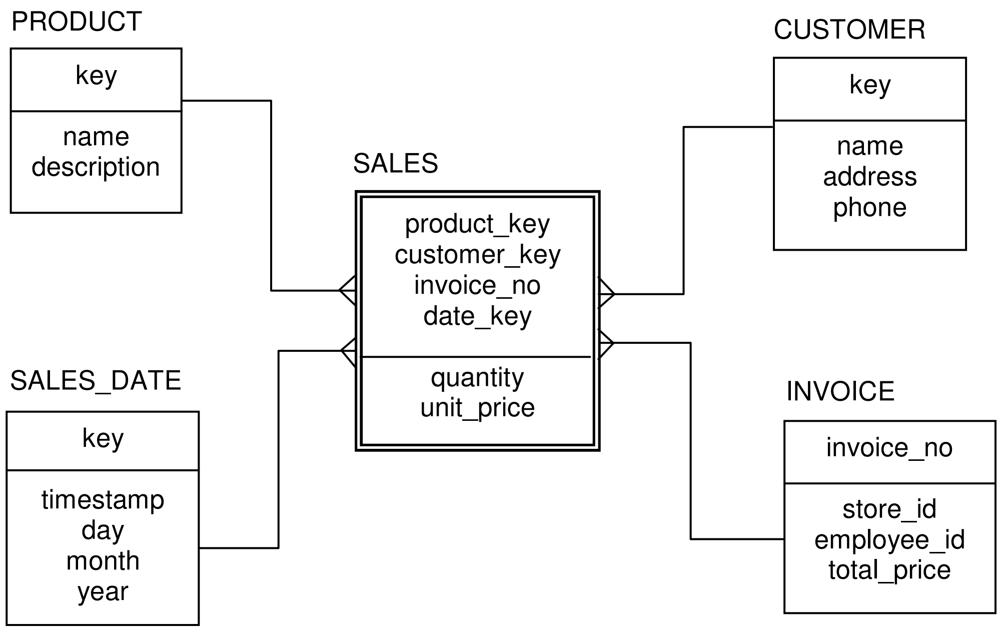


Figure 4.4: **A star schema modeling sales information**: The primary key of the fact table `SALES` is formed by concatenating the foreign keys which point to the dimension tables. A query profile notes `SALES` is the target of many queries with restrictions that cite the foreign key columns. These columns are therefore indexed and targeted for further analysis. Refer to **Example 4.4.1**.

profile for more analysis. This is our rule discovery phase. Currently, this phase is highly restricted and is not subject to the problem of "exponential explosion" described by, for example, (Sun & Yu 1994, Sciore & Siegel 1990, Shenoy & Ozsoyoglu 1989, King 1981). We perform two simple types of rule discovery on table columns identified as being restricted in simple queries or joins, or that form the join columns in equi-joins:

- For continuous data, we collect minimum and maximum values (e.g. if the target column stores real numbers). For discrete data, we collect the distinct values (e.g. if the target column stores the five string values `A,B,C,D,E`).
- We perform a limited search for data holes on a subset of columns which are judged (from the query profile) as being "important" in that they frequently appear in query restrictions or as join columns[14]. Each target column is analyzed to find the N largest gaps in the data, where $N = 1, 2, 3, \ldots$ and is typically less than 10. This is restricted to columns of type numeric or date. We describe the search for data holes in more detail below in Section 4.4.2.

[14]We reiterate that this type of judgment is already commonly made as part of normal DBA duties. Columns which are frequently cited in query restrictions or are join columns are typically candidates for indexing. This was described in Section 2.3.3, page 28.

The motivation for both these data analysis activities is to increase the probability of detecting unsatisfiable queries (Definition 2.2.6, page 18) and zero queries (Definition 4.3.2, page 104). We emphasize the rule discovery phase need not be restricted to the two simple procedures described above. Other types of analysis may be applied such as clustering (see Example 4.4.5, page 114 below). The search for useful semantic rules is quite independent of other functions performed by the semantic optimizer and we report the two strategies above because they are effective but extremely simple to implement.

4. **Monitor Queries**: We perform a limited type of query driven rule discovery[15]. Currently this is restricted to monitoring zero queries. Queries which return no rows are flagged and their restrictions noted. This information is accumulated and used to enhance the semantic information for the cited column. This is explained in detail below in Section 4.4.2.

4.4.2 Utilising Data Holes

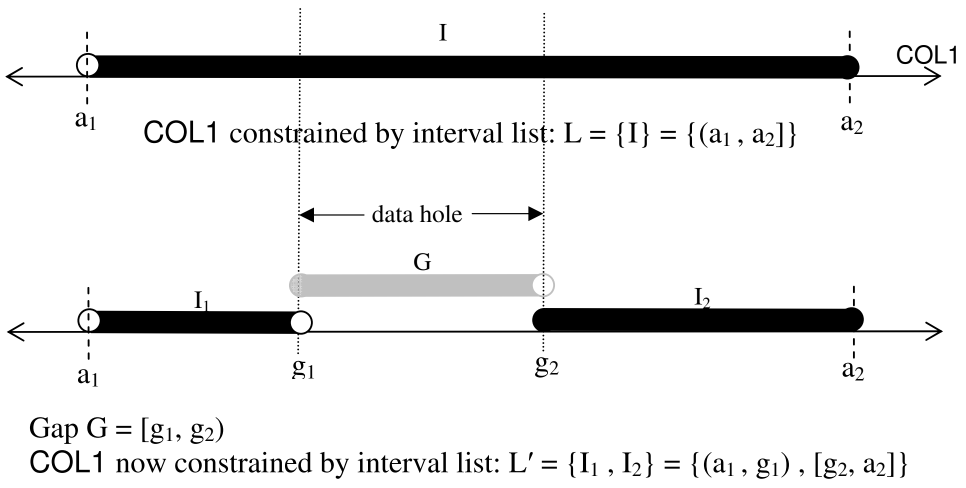

I
COL1
a_1
COL1 constrained by interval list: L = {I} = {(a_1 , a_2]}
a_2
data hole
G
I_1
I_2
a_1
g_1
g_2
a_2
Gap G = [g_1, g_2)
COL1 now constrained by interval list: L′ = {I_1 , I_2} = {(a_1 , g_1) , [g_2, a_2]}

Figure 4.5: **Finding data holes**: This figure depicts the legal range of values a column variable COL1 may assume. In this example the legal range of values is described by an interval list L consisting of a single interval I. Suppose it is subsequently discovered that a gap in the data exists within this range, described by the interval $G = [g_1, g_2)$. Then removing this gap from interval I results in a new interval list L' consisting of two intervals I_1 and I_2. Refer to **Example 4.4.2**.

We now explain why collecting information about data holes in a target column allows us to refine the semantic information we have about that column. In Section 4.4.1 above, we described two techniques to collect information about data holes, one data driven and the other query driven. In both cases the motivation is to increase the probability of detecting zero queries. This in turn is motivated by the

[15]See Section 2.3.1 (page 21) for a detailed description of query driven rule discovery.

observation that exactly the same advantage can be gained as for the detection of unsatisfiable queries; i.e., such queries need not be submitted to the database at all.

However information about data holes is collected, this is used to modify the meta-data held for the target column. We emphasize that the extra semantic knowledge we hold about a column (additional to the meta-data already stored and maintained by RDBMS) is captured in just one form, the interval list. Typically we begin by recording only the minimum and maximum values for that column. Then this information is progressively enhanced as further information about data holes is discovered.

Example 4.4.2. *Consider Figure 4.5, which depicts the legal range of values that column variable* `COL1` *may assume. This information was discovered by noting the minimum and maximum values for the column. This range is described by interval list L consisting of a single interval I. It is subsequently discovered that a gap G exists between $[g_1, g_2)$. Removing this gap from interval list L results in a new interval list $L' = \{(a_1, g_1), [g_2, a_2]\}$. We arrive at the new interval list L' by noting that, logically, the gap G must be removed from the original interval list L. That is, we find the conjunction of L with the negation of G:*

$$L' = L \cdot \neg G$$

In practice then, the progressive enhancement of semantic information about a target column proceeds by successive application of the negation algorithm for intervals (Algorithm 3.9.1, page 74) and the conjunction algorithm for interval lists (Algorithm 3.13.2, page 84).

We now consider in more detail the data driven and query driven search for data holes.

4.4.2.1 Data driven search for data holes

In our data driven search for data holes, each target column is analyzed to find the N largest gaps in the data, where $N = 1, 2, 3, \ldots$ and is typically less than 10. This is currently restricted to columns of type numeric or date and to a small subset of columns deemed to be of particular interest. We make the assumption this analysis may be performed "offline" so it does not negatively impact query evaluation. Although this activity is currently not automated and individual target columns are chosen manually, we argue this technique shows considerable promise, for the following reasons.

- It is simple. In the case of numeric and date data, we may calculate the gap directly by considering the distance between successive data items in a sorted list of data.

- It is naturally limited by the application of the simple heuristic to search only for the first few maximal gaps. Ultimately, the decision as to how many gaps to utilise is a trade-off between refining the semantic information held about a target column and the increasing complexity of the interval list that results from the progressive application of that knowledge.
- It is independent of any dimensional knowledge. We target only single columns and retrieve information about gaps in that column alone, in contrast to (Rishe et al. 2003) who search specifically for empty rectangles in two dimensional data. We argue our approach provides maximum flexibility since the semantic information we accumulate is independent of any particular query form or syntax.
- Partial knowledge is useful. Suppose the table containing the target column is so large that sorting the entire column is impractical. In this case, a statistical sample of the table data can be taken, data holes detected and then checked to ensure the ranges really are empty.

4.4.2.2 Query driven search for data holes

Queries which return no rows are flagged and their restrictions noted. This information is accumulated and used to enhance the semantic information for the cited columns.

Example 4.4.3. *Consider a table `TAB` which includes columns `COL1` and `COL2`. For column `COL1`, a numeric column containing continuous data, we begin with knowledge only of the minimum and and maximum values and this is captured by interval list L_1:*

$$L_1 = \{[0, 500]\}$$

For `COL2`, a string column containing discrete data, we begin with a knowledge of the distinct values and this is captured by interval list L_2:

$$L_2 = \{[A, A], [B, B], [C, C], [D, D], [E, E], [F, F], [G, G]\}$$

We pose the following two queries, both of which are satisfiable but nevertheless return no rows; i.e., they are zero queries.

1.
```
select *
from   TAB
where  COL1 >= 100 and COL1 < 400;
```

2.
```
select *
```

```
from   TAB
where  (COL1 > 350 and COL1 <= 500)
or     COL2 in ('A','G');
```

From query 1 *we may conclude the interval* $[100, 400)$ *represents a hole for* `COL1`*. From query* 2 *we may conclude the interval* $(350, 500]$ *represents a hole for* `COL1` *and also that the intervals* $[A, A]$ *and* $[G, G]$ *are holes for* `COL2`*.*

After the first query, L_1 *may be modified to become:*

$$L_1 = \{[0, 100), [400, 500]\}$$

After the second query, L_1 *may be modified to become:*

$$L_1 = \{[0, 100)\}$$

while L_2 *may be modified to become:*

$$L_2 = \{[B, B], [C, C], [D, D], [E, E], [F, F]\}$$

It is possible that the accumulation, in this query driven manner, of knowledge about data holes might result in an interval list consisting of many narrow intervals. This can be controlled by the application of a simple heuristic such as:

- Limit the number of intervals comprising the interval list to some small number (say 10).

- Accumulate “gap knowledge” only in the case where the width of an existing gap is increased.

Currently, our query driven search for data holes proceeds “offline” and is somewhat contrived in that we are not limited by processing times and we have prior knowledge of both our data distribution and range of query restrictions. However, as our empirical results in Chapter 6 confirm, our current implementation is well able to cope with ten or more intervals in a single interval list semantic description of a target column.

Although it is beyond the scope of this thesis, we predict that the accumulation of knowledge about data holes in the manner we set out above, will be particularly effective for sparse data where satisfiable queries are posed with roughly equal probability across the entire range of data.

4.4.3 Harvesting Schema Constraints

We now describe in detail how we use the existing schema semantics to derive rules which can be utilised by our semantic query optimizer. The "check" constraint type is the most useful to our semantic optimizer, but we also employ some other constraint types to produce simple but effective query rewrite.

4.4.3.1 Check constraints

ALL_INTERVAL_LISTS

ID	OWNER	TABLE_NAME	COLUMN_NAME	DATATYPE	ILIST
⋮	⋮	⋮	⋮	⋮	⋮
17	APP_OWNER	SALES	UNIT_PRICE	NUMBER	{ (0,5000] }
28	APP_OWNER	CUSTOMER	CUST_CODE	VARCHAR2	{ ['A','A'] , ['B','B'] , ['C','C'] }
⋮	⋮	⋮	⋮	⋮	⋮

Check constraint on SALES.UNIT_PRICE :

```
check (UNIT_PRICE > 0 and UNIT_PRICE <= 5000);
```

Check constraint on CUSTOMER.CUST_CODE :

```
check CUST_CODE in ('A','B','C');
```

Figure 4.6: **Harvesting check constraints**: Check constraints may be converted into an interval list form. This meta-data is stored in table `ALL_INTERVAL_LISTS` which is accessed by the semantic query optimizer. Refer to **Example 4.4.4**.

Commercial RDBMS allow check constraints to be associated with particular columns of a target table. Their purpose is to check that new data inserted into the target column conforms to a rule (the constraint) which must be a boolean sentence that evaluates to `TRUE`, `FALSE` or `null`[16]. We restrict slightly the type of boolean sentence that can be utilised by our optimizer.

- The sentence may only contain references to the target column variable itself[17].
- The check constraint must be able to be expressed as an interval list.

With regard to the first point above, there is one important exception. Our optimizer

[16]All commercial RDBMS implement a three value boolean logic system using the two boolean truth values `TRUE` and `FALSE` with the addition of the value `null`. This subject is beyond the scope of this thesis but does not compromise the points made above.

[17]In the Oracle RDBMS, check constraints may also reference the other column variables in the same row.

is able to utilise *implications*; i.e., check constraints of the form:

$$\neg P_{c_i} \textit{ or } Q_{c_j}$$

where P_{c_i} is some boolean sentence concerning column c_i and Q_{c_j} is some boolean sentence concerning column c_j and both columns c_i and c_j belong to the same table. With regard to the second point above, this does *not* constitute an additional limitation because the interval list is effectively a disjunctive normal form (Pohl & Shaw 1986); i.e., *any* boolean check constraint with one variable can ultimately be converted into the interval list form.

Example 4.4.4. *Reconsider the check constraint of Example 4.2.2 (page 99) which (at data insert time) restricts to a sensible range the value of the column* `UNIT_PRICE` *in table* `SALES`*. Similarly, consider a check constraint which restricts the value of column* `CUST_CODE` *in table* `CUSTOMER` *to the values* `A`*,* `B` *or* `C`*. Figure 4.6 illustrates how we convert the check constraints into an equivalent interval list. This meta-data is stored in table* `ALL_INTERVAL_LISTS` *which is accessed by the semantic optimizer.*

Recall that check constraints are applied only at data insert and update time. Our optimizer effectively applies the constraint at query time. This is a crucial difference. Since the constraint represents a statement about the target column which is *always true*, the semantic optimizer may apply the conjunction rule[18] to the constraint interval list and whatever constraint on the target column appears in the query. This is how the semantic optimizer detects, for example, unsatisfiable queries. The following example shows how we combine the results of different phases of the harvesting of schema semantics.

Example 4.4.5. *Reconsider Example 4.2.2 (page 99). Consider column* `UNIT_PRICE`*. Suppose that in addition to the check constraint, a simple analysis reveals that the minimum price stored is in fact* \$2.99 *while the maximum price is* \$1750.00*. Furthermore, the application of a simple clustering algorithm has revealed the prices fall naturally into three main groups:* 2.99-49.99*,* 75.00-399.99 *and* 650.00-1750.00*. We therefore have three sources of semantic information concerning column* `SALES.UNIT_PRICE`*, each of which converts readily to an interval list. Since all three statements must be true, we combine them by applying the conjunction rule. We proceed in three steps:*

1. *Harvest the check constraint on column* `UNIT_PRICE` *to produce interval list*

[18]See Section 3.13 (page 82) for a precise description of the conjunction rule.

L_1:

$$L_1 = \{(0, 5000]\}$$

2. *Find the minimum and maximum values for column `UNIT_PRICE` to form interval $I = [2.99, 1750.00]$. Apply the conjunction rule with L_1 to form new interval list L_2:*

$$\begin{aligned} L_2 &= con\,(L_1, I) \\ &= con\,(\{(0, 5000]\}, [2.99, 1750.00]) \\ &= \{[2.99, 1750.00]\} \end{aligned}$$

3. *Form the results of the clustering into an interval list C. Apply the conjunction rule with L_2 to form new interval list L_3:*

$$\begin{aligned} C &= \{[2.99, 49.99]\,, [75.00, 399.99]\,, [650.00, 1750.00]\} \\ L_2 &= \{[2.99, 1750.00]\} \\ L_3 &= con\,(L_2, C) \\ &= con\,(\{[2.99, 1750.00]\}, \{[2.99, 49.99]\,, [75.00, 399.99]\,, [650.00, 1750.00]\}) \\ &= \{[2.99, 49.99]\,, [75.00, 399.99]\,, [650.00, 1750.00]\} \end{aligned}$$

As each stage of the analysis proceeds, the resulting interval list is stored as meta-data in table `ALL_INTERVAL_LISTS`. Each of the steps described above resulted in further refinement of the semantic information obtained. Note that we may stop the analysis of the target column at any time if, for example, the analysis is judged to be too computationally intensive.

In the next example we illustrate how meta-data stored in table `ALL_INTERVAL_LISTS` is utilised by the semantic optimizer to recast the original SQL query into a more efficient query.

Example 4.4.6. *Reconsider Example 4.4.5 above. Suppose we wish to know how many sales there are for products whose price is between* \$50.00 *and* \$60.00. *The following SQL query is posed:*

```
select count(1)
from   SALES s
where  s.UNIT_PRICE between 50.00 and 60.00;
```

Under normal circumstances this query will be sent to the database and, because of the large size of the target table `SALES`, require substantial resources to answer. However, the semantic optimizer preprocesses this query, performing the following steps:

1. *Retrieve meta-data from `ALL_INTERVAL_LISTS` pertaining to the column restrictions, if it exists. In this case, the interval list L_1 is retrieved:*

$$L_1 = \{[2.99, 49.99], [75.00, 399.99], [650.00, 1750.00]\}$$

2. *If an interval list was retrieved, rewrite the column restriction as an interval list. In this case, the interval list L_2 is the result:*

$$L_2 = \{[50.00, 60.00]\}$$

3. *Find the conjunction of the two interval lists L_1 and L_2:*

$$\begin{aligned} con(L_1, L_2) &= con(\{[2.99, 49.99], [75.00, 399.99], [650.00, 1750.00]\}, \{[50.00, 60.00]\}) \\ &= \{\} \end{aligned}$$

The conjunction of the two interval lists yields null. This is an unsatisfiable query which will return no rows. This query need not be submitted to the database.

We complete the description of how we utilise check constraints by summarising in Figure 4.7 the steps taken by the semantic optimizer to preprocess SQL queries.

4.4.3.2 Cost of Semantic Preprocessing

We now describe how the costs incurred in semantically preprocessing queries arise. We first consider the cost of preprocessing queries then consider the extra cost of processing joins in an analogous manner.

1. *Query preprocessing cost*: Consideration of Figure 4.7 above reveals why semantically preprocessing queries is not costless. For each query restriction, we must
   - search for the relevant interval list L_c
   - convert the restriction into an interval list R_c
   - find the conjunction of L_c and R_c
   - convert the conjunction back to an SQL restriction
2. *Join preprocessing cost*: When joins are preprocessed in an analogous manner, the following *extra* costs arise.
   - Suppose the join clause is "`where t1.COL1 = t2.COL3`". This implies that whatever restrictions apply to "`t1.COL1`", we can also apply these to "`t2.COL3`" and *vice versa*. So before any join restrictions are considered, we already must search for two interval lists and find one conjunction.

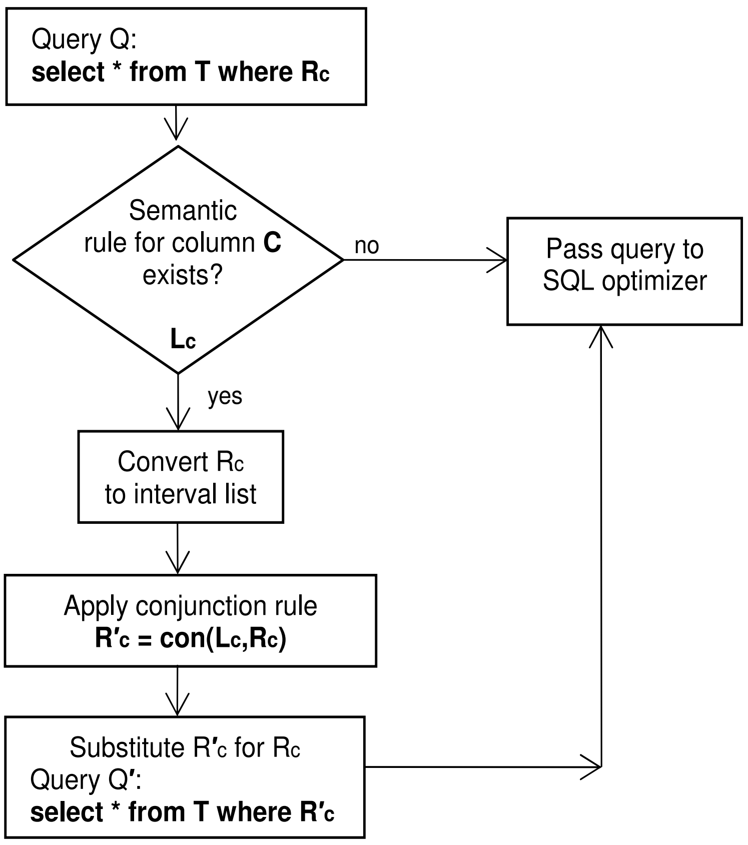

Query Q:
select * from T where Rc
Semantic rule for column C exists?
Lc
no
Pass query to SQL optimizer
yes
Convert Rc to interval list
Apply conjunction rule
R′c = con(Lc,Rc)
Substitute R′c for Rc
Query Q′:
select * from T where R′c

Figure 4.7: **Utilising check constraints**: The semantic query optimizer can preprocess SQL queries where a semantic rule exists for the column (C) cited in the query restriction. Such a rule is always true, so we may find the conjunction of the interval list representing the check constraint (L_c) and the query restriction (R_c). The result of the conjunction (R'_c) is substituted for the original restriction. This is how, for example, we detect unsatisfiable queries.

- For each join restriction, we carry out the same steps set out above for queries.
- We must consider the special case when one of the join restrictions cites one of the join columns. Suppose the join clause is "`where t1.COL1 = t2.COL3`" and one of the restrictions is "`and t1.COL1 < 10`". But this restriction must logically also apply to "`t2.COL3`". Therefore an extra conjunction is required.

In general, semantically preprocessing equi-joins incurs approximately four times the cost of semantically preprocessing queries.

4.4.3.3 Primary key and unique key constraints

When primary key and unique key columns appear in the *select* clause of the SQL query, we make the disarmingly simple change of removing the redundant key word "`distinct`", if it exists. This small change increases the efficiency of the query because it removes the necessity to sort the query results and remove duplicates when in fact we know *a priori* that no duplicates can exist[19]. In the following examples, we show how the Oracle RDBMS itself judges the cost of the queries by requesting the actual execution plan the SQL optimizer chooses to answer the queries. The target tables, the queries and the execution plans in all of the examples for the remainder of this Section are all real and were carried out on the same database and under the same conditions as for the empirical results we report in Chapter 6[20].

Example 4.4.7. *We pose two simple SQL queries, one with the (redundant) keyword "`distinct`" and one without. The target table in this example, `TAB5`, contains* 1×10^6 *rows and occupies* 2.8*Gb of disk storage. Column `ID` is the primary key of `TAB5`, guaranteeing the uniqueness of each value retrieved.*

```
select distinct t5.ID
from   TAB5 t5
where  t5.COL1 > t5.COL2;
```

This query returned 1641 *rows and the SQL optimizer chose the following execution plan. We are concerned primarily with the total cost and total time which are the top figures in the `Cost` and `Time` columns respectively*[21].

```
----------------------------------------------------------------------------
| Operation                | Name         | Rows  | Bytes | Cost  | Time |
----------------------------------------------------------------------------
| SELECT STATEMENT         |              |     1 |       | 12578 | 03:21|
|  SORT AGGREGATE          |              |     1 |       |       |      |
|   VIEW                   |              | 55369 |       | 12578 | 03:21|
|    VIEW                  |              | 55369 |   973K| 12578 | 03:21|
|     HASH JOIN            |              |       |       |       |      |
|      HASH JOIN           |              |       |       |       |      |
|       INDEX FAST FULL SCAN| PK_TAB5      | 55369 |   973K|  2524 | 00:41|
|       INDEX FAST FULL SCAN| NX_TAB5_COL1 | 55369 |   973K|  2979 | 00:48|
|      INDEX FAST FULL SCAN | NX_TAB5_COL2 | 55369 |   973K|  2979 | 00:48|
----------------------------------------------------------------------------
```

We pose the same query without the "`distinct`" keyword:

```
select t5.ID
from   TAB5 t5
where  t5.COL1 > t5.COL2;
```

for which the SQL optimizer produces the following execution plan:

[19]For simplicity, currently we do not allow the occurrence of `null` values in the unique key columns.

[20]The Oracle SQL optimizer, for all results reported this thesis, is set to "cost based". Cost based optimization does not function correctly unless up-to-date statistics exist pertaining to data distribution in the target tables (Chan 2005*d*). It may be assumed that up-to-date statistics exist for all the following examples.

[21]The execution plan printout has been edited for presentation purposes. A more detailed discussion of the format of the SQL optimizer's execution plan and its precise meaning is beyond the scope of this thesis. We utilise this facility here because it clearly indicates the effect of the keyword "`distinct`" on relative query cost.

```
-----------------------------------------------------------------------
| Operation               | Name          | Rows   | Bytes | Cost   | Time   |
-----------------------------------------------------------------------
| SELECT STATEMENT        |               |     1 |     12 |  6650 | 01:46 |
|  SORT AGGREGATE         |               |     1 |     12 |       |       |
|   VIEW                  |               | 55369 |   648K|  6650 | 01:46 |
|    HASH JOIN            |               |       |       |       |       |
|     INDEX FAST FULL SCAN| NX_TAB5_COL1 | 55369 |   648K|  2979 | 00:48 |
|     INDEX FAST FULL SCAN| NX_TAB5_COL2 | 55369 |   648K|  2979 | 00:48 |
-----------------------------------------------------------------------
```

Comparing the total cost: 12578 *versus* 6650 *and total time:* 03 : 21 *versus* 01 : 46 *we see removal of the "*`distinct`*" keyword halves the cost of the query.*

We now give a similar example, but this time we look at an equi-join between two large tables where the join column is in fact the primary key for both tables.

Example 4.4.8. *The target tables in this example,* `TAB5` *and* `TAB6`*, both contain* 1×10^6 *rows and occupy* 2.8*Gb of disk storage each. Column* `ID` *is the primary key for both tables. The simplicity of the following query and the fact that it returns no rows do nothing to mitigate the negative effect of the redundant "*`distinct`*" keyword.*

```
select distinct t5.ID
from   TAB5 t5, TAB6 t6
where  t5.ID = t6.ID;
```

for which the SQL optimizer produces the following execution plan:

```
------------------------------------------------------------------------
| Operation              | Name    | Rows   | Bytes |TempSpc| Cost   | Time |
------------------------------------------------------------------------
| SELECT STATEMENT       |         |  1000K|    11M|       |  4626 | 01:14|
|  HASH UNIQUE           |         |  1000K|    11M|    38M|  4626 | 01:14|
|   HASH JOIN            |         |  1000K|    11M|    17M|   791 | 00:13|
|    INDEX FAST FULL SCAN| PK_TAB5 |  1000K|  5859K|       |   168 | 00:03|
|    INDEX FAST FULL SCAN| PK_TAB6 |  1000K|  5859K|       |   168 | 00:03|
------------------------------------------------------------------------
```

We pose the same query without the "`distinct`*" keyword:*

```
select t5.ID
from   TAB5 t5, TAB6 t6
where  t5.ID = t6.ID;
```

for which the SQL optimizer produces the following execution plan:

```
---------------------------------------------------------------------------
| Operation             | Name    | Rows   | Bytes |TempSpc| Cost   | Time  |
---------------------------------------------------------------------------
| SELECT STATEMENT      |         |  1000K|    11M|       |   791 | 00:13 |
|  HASH JOIN            |         |  1000K|    11M|    17M|   791 | 00:13 |
|   INDEX FAST FULL SCAN| PK_TAB5 |  1000K|  5859K|       |   168 | 00:03 |
|   INDEX FAST FULL SCAN| PK_TAB6 |  1000K|  5859K|       |   168 | 00:03 |
---------------------------------------------------------------------------
```

Comparing the total cost: 4626 *versus* 791 *and total time:* $01:14$ *versus* $00:13$ *we see removal of the "*`distinct`*" keyword reduces the cost by over* 80%. *We include an extra column "*`TempSpc`*" in the execution plans above which indicates the temporary space the SQL optimizer needs to utilise to answer the query. The redundant "*`distinct`*" keyword triggers the appropriation of an extra* 38*Mb of temporary space.*

The objective of presenting Examples 4.4.7 and 4.4.8 above is to highlight the beneficial effect of a seemingly trivial syntactic change to the SQL query. We argue that such a simple rewrite procedure should be implemented in any practical semantic query optimizer.

4.4.3.4 Not null constraints

The purpose of the "not null" constraint on a column is to ensure that only non-null values are ever inserted into that column. This suggests another simple rewrite rule, analogous to the removal of the "`distinct`" keyword described above in Section 4.4.3.3. We propose to remove redundant "`is not null`" restrictions where the cited column already has the "not null" constraint. Again, the proposed rewrite seems disarmingly simple, so perhaps the extra processing is not worthwhile. The following example examines this issue.

Example 4.4.9. *The target table in this example,* `TAB1`*, contains* 2×10^5 *rows and occupies* 539*Mb of disk storage. Columns* `COL1`*,* `COL2`*,* `COL3`*,* `COL4`*,* `COL5` *are all constrained to be non-null, as is the primary key column* `ID`*. The following shows the relevant fragment of the table's definition held by the RDBMS:*

```
Name              Null?    Type
----------------- -------- ----------
ID                NOT NULL NUMBER
COL1              NOT NULL NUMBER
COL2              NOT NULL NUMBER
COL3              NOT NULL NUMBER
COL4              NOT NULL NUMBER
COL5              NOT NULL NUMBER
   .                  .        .
   .                  .        .
```

Furthermore, highly selective indexes exist on all six columns. The following shows index information held by the RDBMS for table `TAB1`*:*

```
Index                STATUS  TABLE_NAME  Columns  distinct  sel%
-------------------- ------- ----------- -------- --------- ------
PK_TAB1              VALID   TAB1        ID       200,000   100.0
NX_TAB1_COL1         VALID   TAB1        COL1     142,172    71.1
NX_TAB1_COL2         VALID   TAB1        COL2     142,162    71.1
NX_TAB1_COL3         VALID   TAB1        COL3     142,008    71.0
NX_TAB1_COL4         VALID   TAB1        COL4     141,933    71.0
NX_TAB1_COL5         VALID   TAB1        COL5     142,161    71.1
```

Now consider the following SQL query against `TAB1` *which returns just* 318 *rows:*

```
select ID
from   TAB1
where  COL2 < COL1
and    COL3 is not null
and    COL4 is not null
```

for which the SQL optimizer produces the following execution plan:

```
------------------------------------------------------------
| Operation         | Name | Rows  | Bytes | Cost  | Time  |
------------------------------------------------------------
| SELECT STATEMENT  |      | 11108 |   325K|  5251 | 01:24 |
|  TABLE ACCESS FULL| TAB1 | 11108 |   325K|  5251 | 01:24 |
------------------------------------------------------------
```

We see the SQL optimizer has opted for a full table scan despite the small cardinality of the answer set and the existence of selective indexes, with a total cost of 5251*. We now pose the same query with the redundant restriction "*`and COL4 is not null`*" removed:*

```
select ID
from   TAB1
where  COL2 < COL1
and    COL3 is not null;
```

for which the SQL optimizer produces the following execution plan:

```
---------------------------------------------------------------------------
| Operation                | Name          | Rows  | Bytes | Cost  | Time  |
---------------------------------------------------------------------------
| SELECT STATEMENT         |               | 11108 |   260K|  4019 | 01:04 |
|  VIEW                    | idx$_jn$_001  | 11108 |   260K|  4019 | 01:04 |
|   HASH JOIN              |               |       |       |       |       |
|    HASH JOIN             |               |       |       |       |       |
|     HASH JOIN            |               |       |       |       |       |
|      INDEX FAST FULL SCAN| NX_TAB1_COL1 | 11108 |   260K|   597 | 00:10 |
|      INDEX FAST FULL SCAN| NX_TAB1_COL2 | 11108 |   260K|   597 | 00:10 |
|     INDEX FAST FULL SCAN | NX_TAB1_COL3 | 11108 |   260K|   597 | 00:10 |
|    INDEX FAST FULL SCAN  | PK_TAB1      | 11108 |   260K|   505 | 00:09 |
---------------------------------------------------------------------------
```

This time the optimizer opts to use various relevant indexes and the cost reduces to 4019*. Finally, we remove the last redundant restriction "*`and COL3 is not null`*":*

```
select ID
from   TAB1
where  COL2 < COL1;
```

for which the SQL optimizer produces the following execution plan:

```
--------------------------------------------------------------------------
| Operation                | Name         | Rows  | Bytes | Cost  | Time  |
--------------------------------------------------------------------------
| SELECT STATEMENT         |              | 11108 |   195K|  2612 | 00:42 |
|  VIEW                    | idx$_jn$_001 | 11108 |   195K|  2612 | 00:42 |
|   HASH JOIN              |              |       |       |       |       |
|    HASH JOIN             |              |       |       |       |       |
|     INDEX FAST FULL SCAN| NX_TAB1_COL1 | 11108 |   195K|   597 | 00:10 |
|     INDEX FAST FULL SCAN| NX_TAB1_COL2 | 11108 |   195K|   597 | 00:10 |
|    INDEX FAST FULL SCAN | PK_TAB1      | 11108 |   195K|   505 | 00:09 |
--------------------------------------------------------------------------
```

Comparing the total costs for the three execution plans, we see removal of the redundant "`is not null`*" restrictions reduces the cost from* 5251 *to* 4019 *to* 2612 *while the total time reduces from* 01 : 24 *to* 01 : 04 *to* 00 : 42*. Removal of the redundant restrictions has halved the query cost.*

The point of the above example is not the details of exactly what execution path is chosen by the optimizer, but the fact that the redundant "`is not null`" clauses have not only provoked the unnecessary checking of data that is already declared to be "not null" in the database's own meta-data but ultimately led to a full table scan despite the presence of relevant selective indexes. We argue any practical semantic optimizer ought to perform this simple query rewrite.

4.4.3.5 Foreign key constraints

The purpose of a foreign key constraint is to maintain referential integrity between parent and child columns. As is the case with all the database constraints we examine in this chapter, the constraint operates only at data insert or update time (if it is enabled) and has no effect whatsoever on queries that cite the foreign key column.

But if all data has been inserted with the foreign key constraint enabled, then every unique value found in the foreign key column is guaranteed to exist in the corresponding primary key column. So any semantic information about intervals that exists for the parent column must also apply to the foreign key column that points to it[22].

We make use of this relationship by automatically copying over to the foreign key column the semantic information we have derived for the parent column. The foreign key column must be *at least* as restricted as the parent column it points to, but it might well be more restricted. If the foreign key column is frequently cited in query restrictions, this is strong motivation for a data analysis on this column.

Example 4.4.10. *Table `CUSTOMER` contains a foreign key column `CITY_CODE` which points to parent column `CODE`, the primary key of reference table `CITY`. Suppose that table `CITY` also has the following check constraint on column `CODE`:*

```
check CODE in ('Auckland','Wellington','Sydney','Melbourne');
```

As we have explained above in Section 4.4.3, this semantic information is harvested and appears as meta-data (in the form of an interval list) in table `ALL_INTERVAL_LISTS` under the entry for `CITY.CODE`. Since column `CUSTOMER.CITY_CODE` is a foreign key pointing to `CITY.CODE`, we are justified in immediately copying over the same semantic information under the entry for `CUSTOMER.CITY_CODE`.

However, a subsequent data analysis of column `CUSTOMER.CITY_CODE` reveals that all values are either `'Wellington'` or `'Melbourne'`. This result is more restrictive than the original, so it replaces the interval list under the entry for `CUSTOMER.CITY_CODE`.

4.5 Conditional Semantic Rules

In this section we describe how the effectiveness of the semantic query optimizer can be enhanced by the addition of simple *conditional* rules. The semantic rules we have described so far are assumed to be *always true*. Therefore, we can utilise them without any pre-conditions. For example, in the case of check constraints which we have converted to a corresponding interval list, we simply apply the conjunction rule with the relevant query restrictions, since both must be true. This is described in detail above in Section 4.4 (page 104).

We now show how the interval list form we use to store semantic rules allows us to easily write conditional rules which are *sometimes true* and which may be used

[22]For simplicity, we do not consider the case where the foreign key column is allowed to be `null`. When this is the case, we cannot strictly say the foreign key column inherits *all* of the parent constraints since one of the parent constraints must, by definition, be "`not null`".

to rewrite the original SQL query. We begin with an example which illustrates how the knowledge of a human domain expert may be captured and encoded into a rule which is able to be utilised by the semantic optimizer.

Example 4.5.1. *Consider the schema fragment of Figure 4.4 (page 108). An entire range of products has been retired and are no longer available for sale as of 30 June 2006. A domain expert notes the obsolete products have product codes which fall into three ranges:* `1` *to* `100`*,* `300` *to* `350` *and* `900` *to* `999`*. So it must be the case that if a product falls into one of these three ranges, the sale was made on or before 30 June 2006. The company did not exist before 1 January 2000, so this must be the earliest possible date*[23]*. This information can be encoded into a simple rule using two interval lists. Let P stand for the product codes and D the sales date. Then our rule has the form "if P then D" where:*

$$\begin{aligned} P &= \{[1, 100] \mid [300, 350] \mid [900, 999]\} \\ D &= \{[20000101, 20060630]\} \end{aligned}$$

Now consider a query which asks for total sales of product "55" during the month of August 2006:

```
select count(1)
from   SALES s
where  PRODUCT_KEY = 55
and    DATE_KEY between 20060801 and 20060831;
```

Writing both restrictions as interval lists we obtain:

$$\begin{aligned} P' &= \{[55, 55]\} \\ D' &= \{[20060801, 20060831]\} \end{aligned}$$

Interval list P′ triggers the rule "if P then D" because interval list P subsumes *P′. Therefore the restriction captured by interval list D must also be true and we may apply the conjunction rule to D and D′:*

$$\begin{aligned} con\,(D, D') &= con\,(\{[20000101, 20060630]\}, \{[20060801, 20060831]\}) \\ &= \{\} \end{aligned}$$

The conjunction of the two interval lists yields null. This is an unsatisfiable query which will return no rows. This query need not be submitted to the database.

[23]For simplicity we assume the product codes are all numeric and the date may be represented by an integer of the form "YYYYMMDD".

4.5.1 Meaning of a Conditional Rule

We now clarify exactly what is meant by a conditional rule of the form:

$$if\ L_{c_i}\ then\ L_{c_j}$$

where L_{c_i} and L_{c_j} are interval lists. These interval lists are statements concerning the allowed values of particular columns c_i and c_j respectively, of some table T. Recall from Chapter 3 that an interval list is simply a shorthand way of writing a sentence in first order logic which constrains the values of a variable to be within certain ranges.

Therefore, when we write "*if* L_{c_i} *then* L_{c_j}" we mean precisely that if the values of column c_i fall within the ranges allowed by L_{c_i}, it must also be the case that the values of column c_j fall within the ranges allowed by L_{c_j}.

4.5.2 Meeting the Condition: the Subsumption Rule

Example 4.5.1 above illustrates how we decide if the pre-condition for a conditional semantic rule is met by a query restriction. We simply note if the query restriction is *subsumed* by the rule pre-condition. If it is, we may add the right hand side of the conditional rule to the query as an additional restriction. This is ultimately a consequence of Theorem 3.16.3 (page 92). We now state this relationship precisely as the Subsumption Rule.

Theorem 4.5.1. ***The Subsumption Rule****:*

- *Let c_i and c_j be the i^{th} and j^{th} columns respectively of table T.*
- *Let L_{c_i} be an interval list describing a range of allowed values for c_i.*
- *Let L_{c_j} be an interval list describing a range of allowed values for c_j.*
- *Let C be a conditional rule of the form:* `if` $\mathtt{L_{c_i}}$ `then` $\mathtt{L_{c_j}}$.

Now consider a query Q which includes a restriction R_{c_i} on column c_i. Then the Subsumption Rule is:

$$If\ L_{c_i}\ subsumes\ R_{c_i}\ then\ replace\ R_{c_i}\ by\ \left(R_{c_i}\ and\ L_{c_j}\right).$$

Proof*: This follows directly from the fact that if L_{c_i} subsumes R_{c_i} then R_{c_i} must logically imply L_{c_i} (Theorem 3.16.3). But L_{c_i} in turn logically implies L_{c_j}. Therefore, by the extended syllogism rule of Boolean Algebra (Pohl & Shaw 1986) R_{c_i} logically implies L_{c_j}.*

4.5.3 Utility of Conditional Rules

We use conditional rules to implement *restriction introduction* (Section 2.5.3 page 34) and *restriction removal* (Section 2.5.2, page 31). Currently no commercial RDBMS allows such rules to be captured for the purposes of query optimization and there is no mechanism available in any commercial RDBMS to add predicates to queries. We argue that such conditional rules are desirable to:

- capture the knowledge of domain experts which might not otherwise be utilised;
- capture the results of a mechanical analysis of data which specifically searches for correlations between the data values of different columns in a table.

The second item is typically the scenario that researchers in SQO have in mind when restriction introduction and removal is discussed (Lowden & Robinson 2002, Cheng et al. 1999, Lee et al. 1999). This raises the question as to how adding an additional predicate to an SQL query (as opposed to simplifying the query) could be advantageous. The answer is found in the presence or absence of *indexes* on the target columns.

4.5.3.1 Implementing Restriction Introduction

To implement restriction introduction, we look for correlations between a column c_i which has no existing index and a column c_j which *is* indexed, where the unindexed column is the subject of the rule pre-condition. The hope then is that when queries are restricted on column c_i we may add the correlated restrictions on column c_j, provoking the SQL optimizer to use the index on column c_j. The following example illustrates this methodology.

Example 4.5.2. *Consider the schema fragment of Figure 4.4 (page 108). A mechanical search for semantic rules on table `INVOICE` reveals that column `STORE_ID` is highly correlated with the primary key `INVOICE_NO`. A domain expert notes this is not surprising since each different store is issued with non-overlapping ranges of invoice numbers. A simple rule set emerges which relates each `STORE_ID` to a particular range of invoice number:*

-
  ```
  if STORE_ID = 'Auckland'
  then INVOICE_NO between 1 and 1000000;
  ```

-
  ```
  if STORE_ID = 'Wellington'
  then INVOICE_NO between 1000001 and 2000000;
  ```

-
  ```
  if STORE_ID = 'Sydney'
  then INVOICE_NO between 2000001 and 3000000;
  ```

- `if STORE_ID = 'Melbourne'`
  `then INVOICE_NO between 3000001 and 4000000;`

These rules are easily converted into interval lists and entered as conditional rules for the semantic optimizer to use. Now when `STORE_ID` is cited in the query restrictions, the corresponding range of invoice numbers is added as an additional restriction. This increases query efficiency because `INVOICE_NO` is indexed while `STORE_ID` is not.

4.5.3.2 Implementing Restriction Removal

To implement restriction removal, we look for correlations between a column c_i which has an existing index of high selectivity and a column c_j which is *not* indexed, where the indexed column is the subject of the rule pre-condition. (This is the converse of the method for restriction introduction described above in Section 4.5.3.1). If the rule pre-condition is met (i.e., L_{c_i} subsumes the query restriction R_{c_i}) then, logically, the rule consequent L_{c_j} could be added to the query without changing the query result. However, this additional restriction itself is unlikely to optimize the query since the column c_j is unindexed. In this case our objective is to *eliminate* a query restriction on column c_j, say R_{c_j}. This is logically permitted if R_{c_j} can be implied by the rule consequent L_{c_j} (i.e., R_{c_j} subsumes L_{c_j}). The following example illustrates this methodology.

Example 4.5.3. *Reconsider the schema fragment of Figure 4.4 (page 108). A mechanical search for semantic rules on table `INVOICE` is carried out in the converse sense to that described above in Example 4.5.2. This time, `INVOICE_NO` forms the rule pre-condition and it is found to be highly correlated with column `STORE_ID`. A simple rule emerges:*

- `if INVOICE_NO between 1 and 1000000`
  `then STORE_ID = 'Auckland';`

The following query is posed:

```
select *
from   INVOICE
where  INVOICE_NO between 500 and 600
and    STORE_ID in ('Auckland','Melbourne');
```

Semantic optimization proceeds in two steps:

1. *The rule pre-condition "`if INVOICE_NO between 1 and 1000000`" subsumes the query restriction "`INVOICE_NO between 500 and 600`" so the rule consequent "`STORE_ID = 'Auckland`" can logically be added to the query.*

2. *However, the query restriction "`STORE_ID in ('Auckland','Melbourne')`" is implied by "`STORE_ID = 'Auckland'`". Therefore it can be eliminated.*

We complete the description of how we utilise *conditional* rules by summarising in Figure 4.8 the steps taken by the semantic query optimizer to preprocess SQL queries.

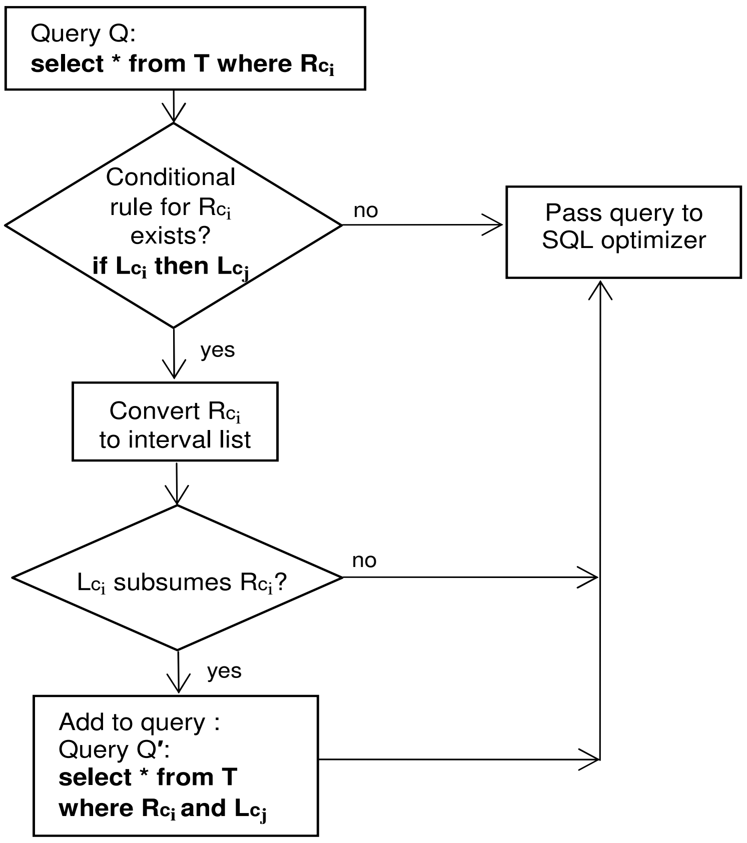

Query Q:
select * from T where Rc$_i$
Conditional rule for Rc$_i$ exists?
if Lc$_i$ then Lc$_j$
no
Pass query to SQL optimizer
yes
Convert Rc$_i$ to interval list
Lc$_i$ subsumes Rc$_i$?
no
yes
Add to query :
Query Q':
select * from T
where Rc$_i$ and Lc$_j$

Figure 4.8: **Utilising conditional rules**: The semantic query optimizer can preprocess SQL queries where a conditional semantic rule exists for the column cited in the query restriction. If the rule pre-condition (left hand side) subsumes the query restriction, the right hand side of the rule may be added to the query as an additional restriction. Typically the rule pre-condition restricts an unindexed column while the rule right hand side restricts an indexed column.

4.6 Summary

In this Chapter we have described the design of a practical semantic query optimizer. We summarise in Figure 4.9 the rules we propose to use for our practical semantic query optimizer.

The main contributions of this Chapter include the following.

- We highlight an intrinsic limitation of SQO in that it depends on the detection of queries which are anomalous. But if anomalous queries are hardly ever

Rule Source	Preparation	Triggered By	Action	Rule Type
Check constraint on column c_i	Convert to interval list Lc_i and store	Column c_i cited in query restriction Rc_i	Substitute Rc_i with con(Lc_i,Rc_i)	Always true
Primary key (PK) or Unique key (UK) constraint on column c_i	nil	PK or UK cited in "select" clause with keyword "distinct"	Delete redundant "distinct" keyword	Always true
Foreign key (FK) constraint on column c_i	Inherit all constraints for corresponding parent column	Column c_i cited in query restriction Rc_i	Substitute Rc_i with con(Lc_i,Rc_i)	Always true
Not null constraint on column c_i	nil	Column c_i cited in query restriction with "is not null"	Delete redundant restriction "c_i is not null"	Always true
Data holes in column c_i	Analyze column c_i to locate gaps. Convert to interval list Lc_i and store	Column c_i cited in query restriction Rc_i	Substitute Rc_i with con(Lc_i,Rc_i)	Sometimes true. Must be revalidated on data update.
Conditional rules	Analyze columns c_i and c_j. Make rule of form "if Lc_i then Lc_j"	Column c_i cited in query restriction Rc_i	If Lc_i subsumes Rc_i then add Lc_j as an additional restriction	Sometimes true. Must be revalidated on data update.

Figure 4.9: **Rules utilised by our semantic optimizer**: This table summarises the rules we propose to use for our practical semantic query optimizer. We harvest schema constraints which are true for the lifetime of the schema. We locate data holes so zero queries can be detected. We analyze data to detect correlations between columns in order to produce conditional rules. Rules that depend on data are only sometimes true and must be revalidated if data is updated.

submitted, perhaps the extra effort of semantically optimizing queries is not worthwhile. To our knowledge, this is the first study to specifically highlight this property of SQO (Section 4.2).

- We then introduce four new terms: *query profile*, *zero query*, *positive query* and *data holes* (Section 4.3). We argue that the first step in any effective implementation of SQO should be the discovery of the query profile which can then be used to initiate a highly focused rule discovery phase (Section 4.4.1).

- We propose a new type of semantic query optimization which searches for "data holes" and utilises them to identify zero queries which, in an analogous fashion to unsatisfiable queries, need not be submitted to the database. We describe two practical methods of discovering data holes, one data driven and one query driven (Section 4.4.2).

- We describe how we harvest a subset of existing schema constraints which are already stored as part of the RDBMS and how these are utilised by our semantic optimizer as rules which are "always true" and which can therefore be added at any time to queries without altering the query outcome (Section 4.4.3). We show why the cost of semantically preprocessing equi-joins is approximately four times the cost of semantically preprocessing queries (Section 4.4.3.2).

- We explain how the optimizer may be extended with conditional rules which are derived from a data driven analysis and which typically capture correlations between non-indexed and indexed columns (Section 4.5). These rules may be elegantly expressed as interval lists and are invoked by application of the Subsumption Rule (Section 4.5.2). To our knowledge, this is the first study to utilise intervals or interval lists in this manner.

Chapter 5

Empirical Methodology

5.1 Introduction

In Chapter 2, we described how current SQL query optimizers cannot utilise semantic information to optimize queries. We concluded that a *reasoning engine* was required which takes semantic information as its input and deduces certain conclusions which allow the original SQL query to be recast to another equivalent query which can be answered more efficiently. Then in Chapter 3 we described an *interval algebra* which we use as the basis of a reasoning engine. In Chapter 4 we described a practical semantic query optimizer which utilises the reasoning engine and which implements various types of semantic query optimization.

In this chapter we present the methodology we employ to carry out a series of empirical investigations whose overall aim is to demonstrate the efficacy of our semantic query optimizer. Our experiments simulate real relational database environments. Thus we seek to not only demonstrate SQO in principle, but also to demonstrate that a practical semantic optimizer can readily be built utilising semantic information already available within the relational database environment.

We begin by describing our experimental methodology and explain the difficulty of obtaining consistent, repeatable results with RDBMS that have automatic database maintenance processes and which have large query caches available. We explain why the experimenter must be careful about what is actually being measured in such circumstances. In particular, we explain why it can be naïve and misleading to use *elapsed time* only as the measure of query efficiency.

We precede our empirical results with the development of a simple cost model pertaining to unsatisfiable queries. We explain how the cost model can greatly assist in predicting the circumstances under which SQO becomes worthwhile. We use the cost model to show that it is straightforward to predict an upper bound to the amount of optimization one can expect when queries are pre-processed by a semantic optimizer.

The remainder of this chapter is organised as follows.

- We explain the difficulty of obtaining consistent, repeatable empirical results with RDBMS where automatic maintenance processes may execute at unpredictable times and where large memory caches are available and how we remediate this problem (Section 5.2.2). We describe how we simulate a busy database environment (Section 5.2.3). We explain how judging the cost of a query by elapsed time alone can be misleading and the metrics we employ to judge the *average query cost* (Sections 5.2.4 and 5.2.5).

- We describe a query normal form which reflects the interval list data type we defined in Chapter 3 (Section 5.3.1). We then describe a qualitative method of classifying *query difficulty* (Section 5.3.2).

- We derive a cost model which predicts the relationship between efficiency gain due to semantic optimization and the probability of an unsatisfiable query being submitted (Section 5.4).

- We conclude by listing the main contributions of the Chapter (Section 5.5).

5.2 Experimental Methodology

In this section we describe the methodology we employ throughout our experiments. We begin by explaining how we treat the Oracle RDBMS as a "black box" and the rationale behind this assumption. We then focus on the problems that arise when trying to obtain consistent, repeatable results with RDBMS that have many automatic maintenance processes and which have large query caches available. We explain why the experimenter must be careful about what is actually being measured in such circumstances.

5.2.1 The Oracle RDBMS as a "black box"

There are a very large number of parameters in the Oracle RDBMS that can be altered by the user: for example, *size of SGA*, *Java pool size*, *sort area size*, *data block checksum* (Rich 2005). In practice, it is infeasible to try to control more than a small number of these and in fact Oracle provides default settings which are rarely altered. A key difference between the *Oracle 10* server (which we use throughout our experiments) and older incarnations of the Oracle RDBMS is that most of the key parameters are determined automatically by the Oracle server using built in heuristics (Cyran, Lane & Polk 2005*b*). The most influential user-defined parameter is arguably `SGA_TARGET` (Cyran, Lane & Polk 2005*d*). This can be (roughly) thought of as the amount of RAM set aside for all Oracle processes.

Given the above, it makes sense to configure the Oracle server as little as possible and accept the defaults provided by Oracle. In this way we establish a baseline for all Oracle instances. Furthermore, this is both recommended by Oracle themselves and is considered current "best practice" in the industry. It also enables us to treat the Oracle server primarily as a "black box" thus eliminating the need to consider a large number of extra variables.

5.2.2 Obtaining consistent results

We now explain the difficulty of obtaining consistent, repeatable results with the Oracle RDBMS, as it is typically configured. We explain why the experimenter must be careful about what is actually being measured. In particular, we explain why it can be naïve and misleading to use *elapsed time* only as the measure of query efficiency.

Two factors present in the Oracle RDBMS have the potential to affect the consistency of empirical measurements. These are:

- Various automatic maintenance processes may execute at unpredictable times, utilising CPU and disk resources. This is increasingly the case as more routine maintenance tasks, formally performed manually by the DBA, are replaced by automatic tasks that may be scheduled or triggered within the database system.

- Large query caches mean that parsed SQL queries, along with query results, can remain in memory for long periods of time. Thus the speed with which an SQL query is answered is deeply affected by the queries which have gone before.

With regard to the first item above, the Oracle RDBMS, in common with other major commercial RDBMS, has moved in recent years to implement partial automation of database administration. The commercial motivation for this is clear but there are also positive effects for the DBA in that some routine maintenance tasks are now automatically scheduled or triggered and carried out without human intervention (Fogel & Lane 2006*a*, Cyran, Lane & Polk 2005*e*). Two key areas which are now automatic are: *extent management* and *segment space management*[1]. Both of these reduce the amount of DBA intervention required, particularly as database objects grow large.

With regard to the second item above, the purpose of caching the result of SQL parsing and query results is to enhance the speed with which queries are processed. When an SQL query is made against the database, the database management system first checks to see if the query is the same as one it has recently parsed. If it is judged to be the same (typically because the query is textually identical to a previous query), the existing parse tree is used because this is quicker than a re-parse (Chan 2005*e*). The database management system then checks to see if the data required is still in the data cache; retrieval from memory is far quicker than retrieval from disk (Cyran, Lane & Polk 2005*a*). So the speed with which an individual SQL query is answered is deeply affected by the query context; i.e., what queries have immediately preceded the current query and what tables have been the target of these queries. It is important to note that when table rows are retrieved from disk,

[1]A more complete discussion of these parameters and their significance is beyond the scope of this thesis. However, detailed information about these and other Oracle database parameters can be accessed via Oracle's online documentation: `http://www.oracle.com/pls/db102/db102.homepage`

it is not the case that only those rows which satisfy the query are retrieved. Rather, data is retrieved from disk in *blocks*. In the case of the Oracle RDBMS, data is typically retrieved in 8K blocks. This data is then placed in memory and remains there until it is aged out by further queries (Chan 2005*c*). Therefore, a subsequent query does not have to be identical to a previous query in order to take advantage of cached data. It is sufficient that subsequent queries address data which is proximate to previously queried data[2].

The objective of describing these aspects of commercial RDBMS and the Oracle RDBMS in particular, is to explain why using *elapsed time* only as the measure of query efficiency can be seriously misleading. In reality, the experimenter cannot know beforehand when automatic maintenance processes will run and it is extremely difficult to run identical query batches in such a way that previous batches do not influence the results of subsequent batches. The next section describes our solution to this problem.

5.2.3 Experimental Setup

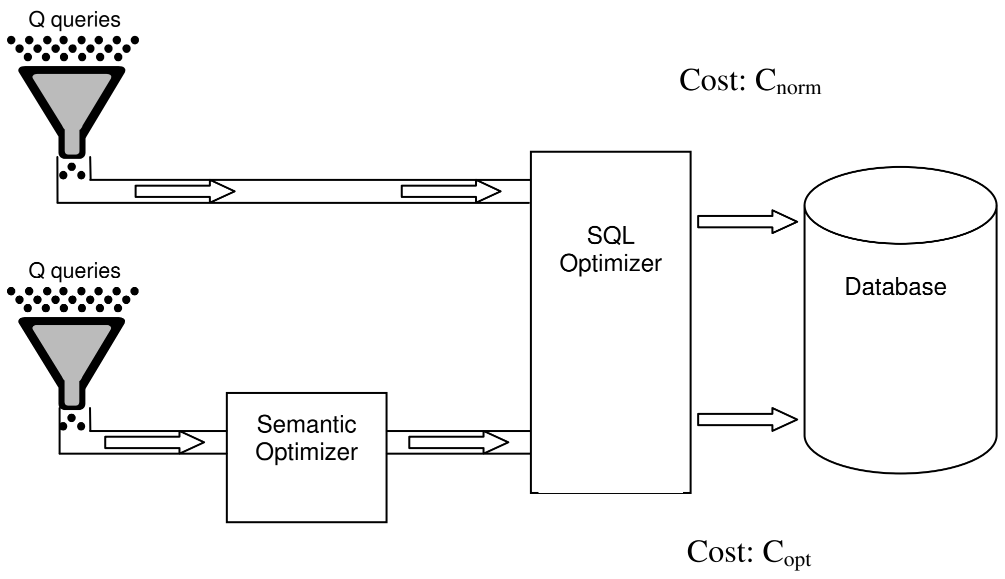


Figure 5.1: **Experimental setup**: Two identically configured query batches are used. One batch runs only semantically optimized queries while the other runs only the identical unoptimized queries. The batches never run together so they never compete for computer resources. We use the Oracle supplied tool `tkprof` to measure the average query cost.

We now describe the experimental setup we use to minimise inconsistency in our empirical results. Refer to Figure 5.1. The main objective of our experiments

[2]This is in fact the motivation for the co-location of table data into *clusters*. In the context of RDBMS, clustering means that tables which are frequently queried together are co-located in the same physical location on disk. This maximises the probability that queried data will already be present in memory, thereby minimising disk activity.

is to demonstrate the efficacy of SQO in certain circumstances. Accordingly, our fundamental experimental activity is to compare the cost of running a large batch of semantically optimized queries against the cost of running those same queries without any semantic optimization. Each of the following subsections focuses on a different aspect of our experimental setup and we explain the importance of each and why we have chosen these particular experimental conditions.

5.2.3.1 Accurately measuring query cost: `tkprof`

We do not simply measure elapsed time in order to judge the cost of a query batch. Rather, we use the Oracle system itself to take its own measurements, which are very precise. We use a software tool, `tkprof`,[3] to take these measurements and this enables us to look at the query cost in a number of different ways. For example, we may look at the CPU time separately from the number of disk blocks physically fetched from disk, or the number of query rows fetched. A detailed description of the measurements we use follows in Section 5.2.4 (page 139) below.

5.2.3.2 Simulating a busy database environment

All our experiments are carried out using *six* target tables, rather than a single table. Each of the six tables has an identical definition and has the same statistical distribution of values in its columns and is the same size (i.e., has the same number of rows) but the table rows are non-identical. Each table has an identical probability of being queried ($\frac{1}{6}$) during the running of any batch and tables are queried in random order. The primary objective of this arrangement is to simulate more closely a busy database environment where a number of tables are being queried, rather than just a single table, as is reported by most experimenters in this area (Lowden & Robinson 2002, Gryz et al. 2001, Gryz, Liu & Qian 1999, Cheng et al. 1999). A secondary objective is to avoid the situation where an entire table is cached at some point in the batch run, thereby systematically distorting the cost measurements.

5.2.3.3 Measuring average query cost

We do not measure the cost of individual SQL queries in the manner reported by (Gryz et al. 2001, Cheng et al. 1999). Rather, we measure the cost of submitting *batches* of many similar [4] queries. Thus our results represent a statistical average

[3]The `tkprof` software tool is well known and heavily employed by Oracle database practitioners to determine, for example, the most costly SQL queries in a batch. It operates by precisely recording the cost of each separate database operation. A more thorough discussion of this tool is beyond the scope of this thesis.

[4]We define precisely what we mean by "similar" queries in Section 5.4 (page 144) below.

which we argue is a better measure of true query cost in that it provides a metric for a whole class of queries, rather than one representative or typical query.

5.2.3.4 Distribution of experimental data

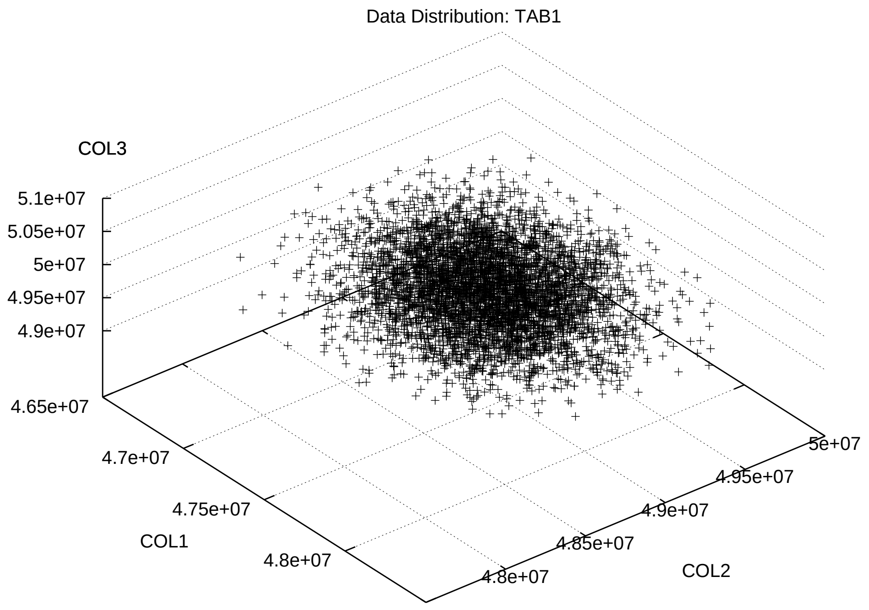


Figure 5.2: **Data distribution**: The above scatter plot depicts data distribution across `COL1`, `COL2` and `COL3` of table `TAB1`. The distribution of each column is a truncated normal distribution where values outside plus or minus three standard deviations are discarded. A similar plot is obtained by plotting any three of columns `COL1` to `COL5` of any of the six tables `TAB1` to `TAB6`.

Although our target tables consist of columns whose data type is a mixture of numeric, string and date (as might be expected in a real world table), for the purposes of measurement we restrict[5] only the first five columns (`COL1` to `COL5`) which are all numeric[6]. The distribution of data values in these columns is a *truncated normal distribution.* That is, values are randomly generated to conform to a normal distribution of given mean and standard deviation, but we discard values beyond plus or minus three standard deviations. We choose this distribution:

- to more realistically simulate real world data. A normal distribution arguably

[5]Precisely, we mean that only columns `COL1` to `COL5` are cited in the SQL query restriction clauses.

[6]There is no loss of generalisation in imposing this restriction since our interval algebra, described in detail in Chpater 3, requires only that the data type has a determinsitic total ordering. Our practical semantic optimizer, described in Chapter 4, is currently able to reason with *numeric*, *string* and *date* data types. However, restricting to numeric data for the purposes of our experiments eases the generation of queries and facilitates comparison with other empirical studies that typically also work with numeric data.

simulates a wider range of actual data distributions than, say, a uniform distribution (Larsen & Marx 1981).

- to facilitate the generation of queries. We use our knowledge of the actual distribution of data to construct different types of queries. This is described below in Section 5.4.1 (page 145).

Data distribution across three columns is pictured in Figure 5.2. Each of the columns `COL1` to `COL5` has a distribution with a different mean but same standard deviation. None of the tables `TAB1` to `TAB6` contains duplicated data; only the statistical distribution is the same.

5.2.3.5 Standardising batch conditions

Before any batch run, the target database instance is closed down and re-started. Furthermore, we specifically empty the cached shared memory resources of the database instance. We then precede the actual query batches with a dummy batch (identical for both normal and optimized batches) whose results are discarded. These actions ensure the query cache is empty at the beginning of each run and then in the same state for both normal and optimized batches before the actual measured batch proceeds. Repeated measurements have shown these preliminary procedures are vital to minimise systematic error accumulating in the results.

5.2.3.6 Indexing restricted columns

Each of the restricted columns, `COL1` to `COL5`, is indexed separately with a standard B-tree index[7]. The primary motivation for indexing in this way is that it is the most likely strategy to be followed in real world database environments for table objects with the characteristics that our six tables share. Indeed, it would be an extraordinary situation in practice (and probably an oversight) that these columns would *not* be indexed. The selectivity of the queried columns is never allowed to fall below the level at which the Oracle SQL optimizer might decide not to consult the appropriate index.

Furthermore, each of the queried columns `COL1` to `COL5` is indexed separately, as opposed to indexing multiple columns in a single index. This ensures that each of the columns is the *leading column*[8] in an existing index. This provides the best

[7]Indexing is a well researched topic and beyond the scope of this thesis. We simply employ the standard Oracle indexing strategy appropriate to the size of the queried tables and the selectivity of the restricted columns.

[8]Only leading columns in an index can be utilised to improve query performance. For example, suppose an index is created on (`COL1`,`COL2`,`COL3`), in that order. Then queries restricted on (`COL1`), (`COL1` and `COL2`), (`COL1` and `COL2` and `COL3`) may all utilise this index. However, queries restricted on, for example, (`COL2`) or (`COL3`) cannot utilise this index.

compromise and the greatest flexibility since the Oracle SQL optimizer is capable of deciding the efficacy of *combining* any of these indexes. For example, suppose a query is restricted on columns `COL1` and `COL4`. The Oracle optimizer is capable of deducing the advantage of combining these indexes, as if the compound index (`COL1`,`COL4`) existed (Chan 2005*b*).

5.2.4 Measuring query cost

In this section we describe three query metrics which we use to measure the true query cost. The metrics we use are all statistics output by the Oracle database tool `tkprof` which we described briefly above in Section 5.2.3. We set out and explain each of these three metrics in Table 5.1.[9] For each of the metrics described in

Metric	**Meaning**	$\mathbf{R_{cost} = \frac{COST_{opt}}{COST_{norm}}}$
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement.	R_{cpu}
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.	R_{elpsd}
DISK	Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.	R_{dsk}
COMBINED	The average of the other three metrics. This metric is only ever reported as a *ratio*.	R_{com}

Table 5.1: **Query cost metrics and their meaning**.

Table 5.1, Oracle further distinguishes between three phases or calls when an SQL statement is processed: `PARSE`, `EXECUTE` and `FETCH`. These are set out in Table 5.2. We typically report the sum of these three calls as a single metric unless we wish to distinguish between the three phases.

We wish to minimise random uncertainties that might influence the outcome of our experiments, such as changing machine load, and to minimise the number of variables we need to consider. To this end, we do not report absolute cost metrics. Instead, we report the *ratio* of the two batch results. For example, when we use

[9]This information is primarily sourced from the Oracle online documentation: http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10752/sqltrace.htm#1018

Call	Meaning
PARSE	Translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns and other referenced objects.
EXECUTE	Actual execution of the statement by Oracle. For INSERT, UPDATE, and DELETE statements, this modifies the data. For SELECT statements, this identifies the selected rows.
FETCH	Retrieves rows returned by a query. Fetches are only performed for SELECT statements.

Table 5.2: **The three SQL statement calls distinguished by analysis tool `tkprof`.**

the metric `DISK`, we employ the ratio of the optimized `DISK` versus the normal `DISK` values, rather than the absolute values themselves. The ratios we report are always of the form $R_{cost} = \frac{COST_{opt}}{COST_{norm}}$ where the numerator $COST_{opt}$ is always the cost measured from the optimized query batch and the denominator $COST_{norm}$ is always the cost measured from the corresponding unoptimized (normal) batch.

5.2.5 Overall Query Cost

Each of the three metrics focuses on a different aspect of computational cost. In order to judge *overall query cost* we combine the three cost metric ratios into one by taking the arithmetic average of the three ratios at each data point. This is reported as the *combined* (`COM`) metric ratio.

Definition 5.2.1. ***Combined Cost Metric Ratio**: R_{com}*

$$R_{com} = \frac{1}{3}\sum_{i=1}^{3}\frac{COST^{i}_{opt}}{COST^{i}_{norm}} \tag{5.1}$$

where $COST^i$ represents each of the three cost metrics described above in Table 5.1.

For simplicity, we do not attempt to weight the individual metric ratios, but judge each individual cost measure as being of equal importance. The combined metric ratio has shown itself to be a remarkably stable measure of overall query cost, across a wide range of experimental conditions. One may observe from the experimental results that while individual cost metric ratios (for example `DISK`: the total number of disk blocks physically read) display considerable variation from the predicted value, it is frequently the case that the combined ratio averages out these individual differences. The metrics that display the most variation are `DISK` (the total number of data blocks physically read from disk) and `ELAPSED` (the total elapsed time). This is to be expected and is discussed above in Section 5.2.2 (page 133).

Cost Metric Ratio	Definition
R_{cpu}	$\frac{CPU_{opt}}{CPU_{norm}}$
R_{elpsd}	$\frac{ELAPSED_{opt}}{ELAPSED_{norm}}$
R_{dsk}	$\frac{DISK_{opt}}{DISK_{norm}}$
R_{com}	$\frac{1}{3}\sum_{i=1}^{3}\frac{COST^{i}_{opt}}{COST^{i}_{norm}}$

Table 5.3: **Cost metric ratio definitions**: We do not report absolute cost metrics. Instead we report the *ratio* of the optimized cost metric to the unoptimized cost metric. The above definitions show how each ratio is defined. The *combined* ratio R_{com} is the average of the other three ratios.

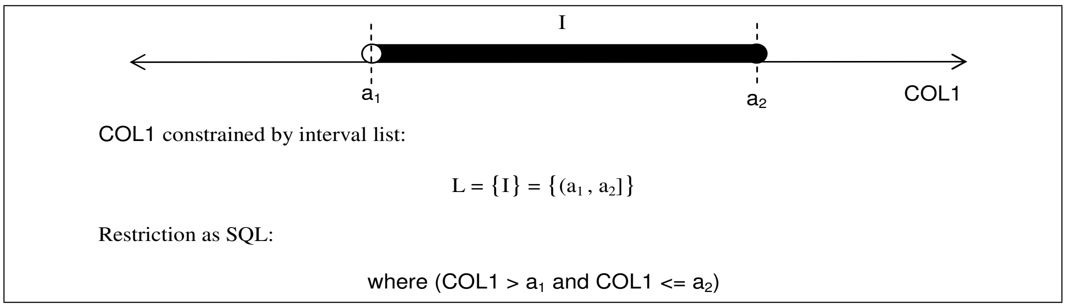
We summarise the ratios we report and their definitions in Table 5.3.

5.3 Query Normal Form and Difficulty

We now describe the form of the queries that make up our test batches and the limitations we place on those queries. Our objective is to restrict the difficulty or complexity of queries we use in our experiments by, for example, disallowing subqueries within the SQL text, but at the same time allowing queries a reasonable expressive power. We study either simple queries with restrictions against a single target table or equi-joins with restrictions against a pair of target tables. All query and equi-join restrictions address numeric columns only. However, we classify the restriction clauses in a novel way. All our restriction clauses may be described by an *interval list* which we introduced in Section 3.8 (page 69). This section explains how the restriction clauses arise naturally from our definition of the interval list.

5.3.1 Query Normal Form

Consider Figure 5.3 which illustrates three query restrictions of increasing complexity. In Figure 5.3(a) we picture an interval list consisting of a single interval. This translates into a simple SQL restriction on column `COL1` consisting of *two* boolean statements representing respectively the left and right bounds of the interval. In Figure 5.3(b) the interval list comprises two intervals, resulting in a more complex restriction on `COL1` consisting of *two pairs* of boolean statements representing re-



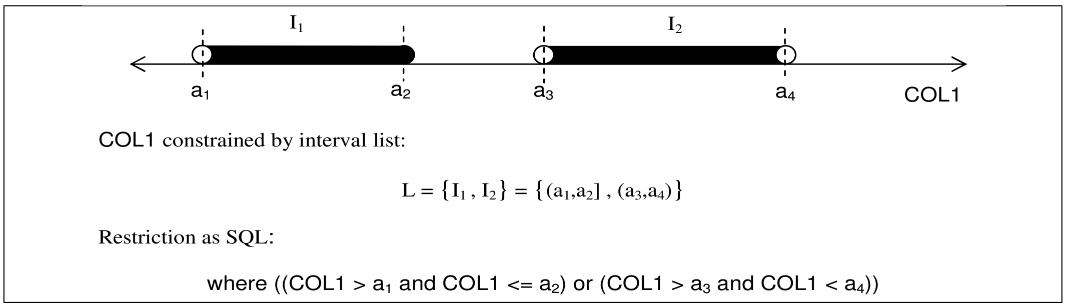
COL1 constrained by interval list:

$$L = \{I\} = \{(a_1, a_2]\}$$

Restriction as SQL:

```
where (COL1 > a1 and COL1 <= a2)
```

(a)



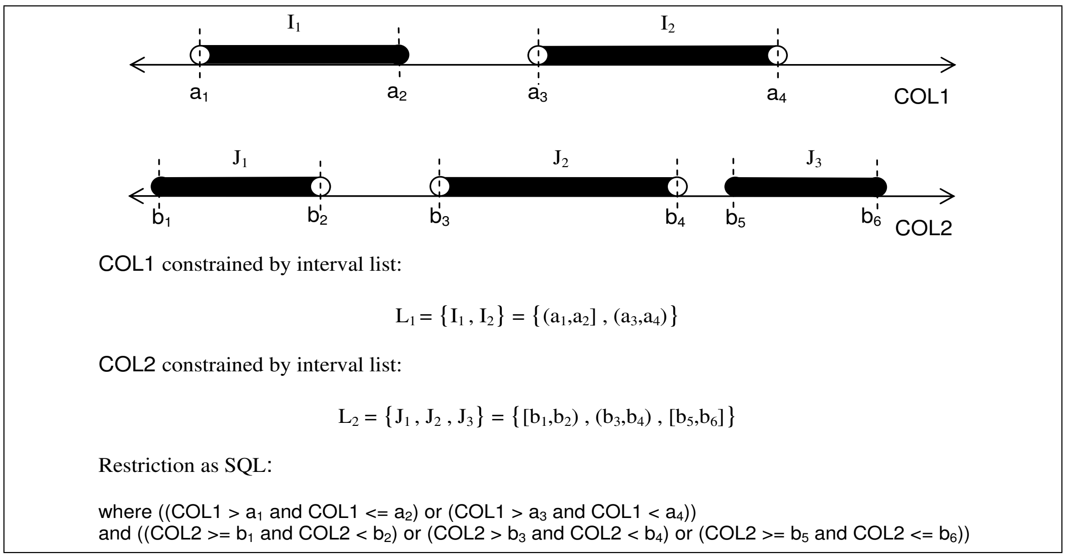
COL1 constrained by interval list:

$$L = \{I_1, I_2\} = \{(a_1,a_2], (a_3,a_4)\}$$

Restriction as SQL:

```
where ((COL1 > a1 and COL1 <= a2) or (COL1 > a3 and COL1 < a4))
```

(b)



COL1 constrained by interval list:

$$L_1 = \{I_1, I_2\} = \{(a_1,a_2], (a_3,a_4)\}$$

COL2 constrained by interval list:

$$L_2 = \{J_1, J_2, J_3\} = \{[b_1,b_2), (b_3,b_4), [b_5,b_6]\}$$

Restriction as SQL:

```
where ((COL1 > a1 and COL1 <= a2) or (COL1 > a3 and COL1 < a4))
and ((COL2 >= b1 and COL2 < b2) or (COL2 > b3 and COL2 < b4) or (COL2 >= b5 and COL2 <= b6))
```

(c)

Figure 5.3: **Depicting interval lists as query restrictions**: Figures 5.3(a) to 5.3(c) illustrate how we map from an interval list into a normal SQL restriction clause. In each case we begin by sketching the interval list which captures the range of values the column may assume. We then rewrite the interval list as a normal SQL restriction clause.

spectively the first and second intervals. In Figure 5.3(c) we picture both column `COL1` and column `COL2` being restricted.

All queries made against our test tables strictly conform to the pattern illustrated in Figure 5.3. We refer to this as our *query normal form*. That is, for both simple queries with restrictions and equi-joins with restrictions, in each case the restriction clauses may be represented by one or more interval lists. The following makes this explicit.

Definition 5.3.1. ***Interval List Restriction (ILR)****: An interval list restriction is an SQL query restriction that is derived from and may be represented by a single interval list.*

We use the acronym ILR to refer to this type of restriction, to distinguish it from the general sense of the term "restriction" in the context of SQL. The simplest ILR is a null restriction, represented by an empty interval list. The simplest non-null ILR is represented by an interval list comprising a single interval and therefore corresponds to *two* boolean statements representing respectively the left and right bounds of the interval. Using the above definition, we now look at the appearance of the SQL queries used in our experiments.

- Simple queries with restrictions against a single target table: All SQL queries of this form have the following pattern.

```
Pattern             Example
<display clause>    SELECT t.COL1, t.COL2, t.COL3
<source clause>     FROM   TAB t
<ILR 1>             WHERE  ((t.COL1 >= 1 and t.COL1 < 25) or
                             (t.COL1 > 50 and t.COL1 < 55) or
                             (t.COL1 >= 100 and t.COL1 <= 200))
<ILR 2>             AND    (t.COL3 > 500 and t.COL3 <= 505);
```

- Equi-joins with restrictions against a pair of target tables. All SQL queries of this form have the following pattern.

```
Pattern             Example
<display clause>    SELECT t1.COL1, t1.COL5, t2.COL7
<source clause>     FROM   TAB1 t1, TAB2 t2
<join clause>       WHERE  t1.COL1 = t2.COL7
<ILR 1>             AND    ((t1.COL5 > 2 and t1.COL5 < 4) or
                             (t1.COL1 >= 10 and t1.COL1 < 50))
<ILR 2>             AND    ((t2.COL8 >= 1 and t2.COL8 < 17) or
                             (t2.COL8 > 25 and t2.COL8 <= 50));
```

In accordance with the definition of an interval list (see Definition 3.8.3, page 71), we insist all intervals comprising an interval list are disjoint; i.e., no interval *overlaps* (Definition 3.7.1, page 65) or *touches* (Definition 3.7.2, page 65) any other. This is illustrated in Figure 5.3 and is reflected in the equivalent SQL restriction clauses. For simplicity, a column variable is referenced by just one ILR; i.e., for any SQL query, each column variable may appear in at most one ILR[10]. In this thesis, we study only cases where the ILRs are joined by the boolean `AND` operator[11].

5.3.2 Query Difficulty

We now describe our method of qualitatively classifying query difficulty. The query normal form described above in Section 5.3.1 suggests two ways of describing the relative difficulty[12] of a query. We consider both the number of ILRs joined by boolean operator `AND` and the number of intervals within each ILR:

- *Vertical difficulty*: The number of ILRs contained in the SQL query. In this thesis, we consider only queries where each interval list restriction is joined by boolean operator `AND`. Thus an SQL query with one ILR is said to have a vertical difficulty of 1 while an SQL query with three ILRs is said to have a vertical difficulty of 3.

- *Horizontal difficulty*: The average number of intervals comprising each ILR. Thus an SQL query with an average of two intervals per ILR is said to have a horizontal difficulty of 2 while an SQL query with an average of five intervals per ILR is said to have a horizontal difficulty of 5.

In our experiments, we vary the relative difficulty of the SQL queries by separately varying both the vertical and horizontal difficulty. Ultimately, we are led to this classification because extending the horizontal difficulty increases the number of intervals comprising an interval list, thereby increasing the number of iterations of the conjunction and disjunction algorithms that are required[13]. Similarly, extending the vertical difficulty increases the number of times the algorithms must be called. In this way, we expect our empirical results to directly reflect the performance of these algorithms.

5.4 Cost Models

In this section we develop several cost models with the objective of quantitatively predicting the amount of optimization we can expect from a semantic query optimizer. We first reiterate definitions for several types of SQL query which we classify

[10]However, in practice, the software used in our experiments will process normally multiple ILRs referencing the same column variable.

[11]However, in practice, the software used in our experiments will process ILRs joined by either `AND` or `OR`.

[12]We use the term "difficulty", rather than "complexity" to avoid ambiguity with "big O", the computational complexity.

[13]See Section 3.13.3 (page 84) and Section 3.12.3 (page 82) for a detailed discussion of the "big O" computational complexity of these algorithms.

in order to understand what query environments are required for semantic optimization to be effective. We then show how our cost models can be used to predict an upper bound for the amount of optimization we can expect for certain types of query environment.

5.4.1 Classifying Queries

We now reiterate definitions for the three types of SQL query which we distinguish by the type of result they produce when submitted to the database. Table 5.4 gives definitions for *positive*, *unsatisfiable* and *zero* queries.

Query Type	**Meaning**
positive	The query returns one or more rows.
unsatisfiable	The query is *logically* excluded from returning any rows because of schema semantics.
zero	The query is unsatisfiable because there is currently no data residing in the database which satisfies the query restrictions (data "holes").

Table 5.4: **Three query types**: The three query types are distinguished by the type of result returned when submitted to the database. We use these three definitions to facilitate the development of our cost models.

5.4.2 Cost Model: Unsatisfiable Queries

We now develop a cost model for SQO in the presence of unsatisfiable queries. We refer to Figure 5.1 (page 135) which depicts our experimental setup and to Table 5.2 (page 140) which describes the three phases of query execution delineated by the analysis tool `tkprof`.

Consider a batch of Q queries, q_u of which are unsatisfiable, which we submit once to the database instance where they will be processed normally (a *normal* batch) and once to the same database instance where they will be pre-processed by our semantic optimizer (an *optimized* batch).

- Let t_{prs} be the average time required by the SQL optimizer to parse each query. This corresponds to the `PARSE` phase described in Table 5.2.

- Let t_{sem} be the average time required by the semantic preprocessor for each query. This applies only to the query batch that is to be semantically optimized.

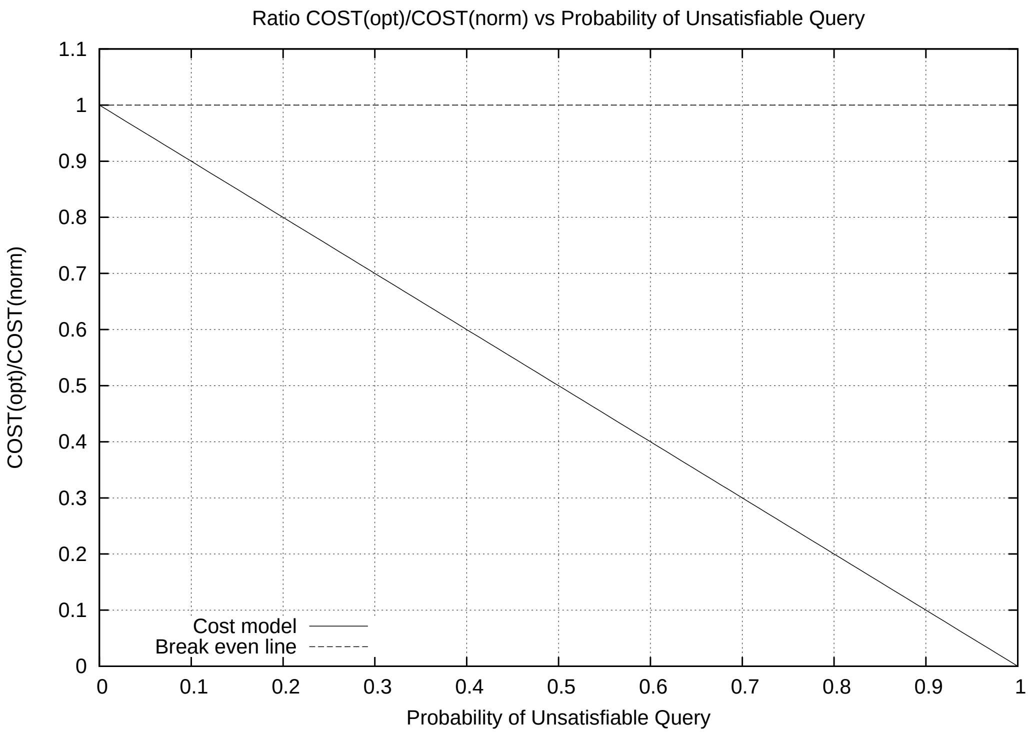


Figure 5.4: **Cost model for unsatisfiable queries**: Our cost model above predicts a straightforward relationship between P_u, the probability of an unsatisfiable query and the ratio of the optimized batch cost $COST_{opt}$ to the normal batch time $COST_{norm}$. Even when t_{sem}, the average time to semantically optimize each query is negligible compared with t_{ora}, the normal time taken to parse, execute and fetch the query, we cannot expect better optimization than indicated by this line.

- Let t_{get} be the average time required to execute and fetch each query. This corresponds to the `EXECUTE` and `FETCH` phases described in Table 5.2.

For the normal batch, the total time required to run the batch, T_1 is given by:

$$T_1 = Q\left(t_{prs} + t_{get}\right) \tag{5.2}$$

For an optimized batch, q_u are unsatisfiable and so need never be submitted to the database. So the total time required to run the batch, T_2 is given by:

$$\begin{aligned} T_2 &= (Q - q_u)\left(t_{sem} + t_{prs} + t_{get}\right) + q_u t_{sem} \\ &= Q\,t_{sem} + Q\,t_{prs} + Q\,t_{get} - q_u t_{sem} - q_u t_{prs} - q_u t_{get} + q_u t_{sem} \\ &= Q\left(t_{sem} + t_{prs} + t_{get}\right) - q_u\left(t_{prs} + t_{get}\right) \end{aligned}$$

Now set $t_{ora} = t_{prs} + t_{get}$. The time t_{ora} corresponds to the three query phases `PARSE`, `EXECUTE` and `FETCH` set out in Table 5.2. We may write:

$$T_2 = Q\left(t_{sem} + t_{ora}\right) - q_u t_{ora} \tag{5.3}$$

The result for the total batch time T_2 given in Equation 5.3 for a semantically optimized batch is completely intuitive: we pay the penalty of the extra processing time t_{sem} in the first term and make the saving from not submitting the q_u queries to the database in the second term.

5.4.2.1 Difference between processing times

Consider the *difference* between the two processing times, $T_2 - T_1$:

$$\begin{aligned} T_2 - T_1 &= Q\,(t_{sem} + t_{ora}) - q_u t_{ora} - Q\left(t_{prs} + t_{get}\right) \\ &= Q\,t_{sem} + Q\,t_{ora} - q_u t_{ora} - Q\,t_{ora} \\ &= Q\,t_{sem} - q_u t_{ora} \end{aligned} \tag{5.4}$$

So, to minimize $T_2 - T_1$ (and preferably make it negative; i.e., a time *saving*) we could minimise the first term on the right by minimising t_{sem}, the average time required to semantically preprocess each query. Similarly, if the second term on the right is large, we might also achieve a time saving. This confirms the intuition that the higher the cost of processing the queries normally, the higher the potential saving. Or, if the proportion of unsatisfiable queries is high, we achieve the same effect.

5.4.2.2 Ratio of processing times

The *ratio* of the two processing times, T_2/T_1 is given by:

$$\frac{T_2}{T_1} = \frac{Q\,(t_{sem} + t_{ora}) - q_u t_{ora}}{Q\,t_{ora}} \tag{5.5}$$

To make a time *saving* (positive optimization), this ratio must be less than 1:

$$\begin{aligned} \frac{Q\,(t_{sem} + t_{ora}) - q_u t_{ora}}{Q\,t_{ora}} &< 1 \\ Q\,(t_{sem} + t_{ora}) - q_u t_{ora} &< Q\,t_{ora} \\ Q\,(t_{sem} + t_{ora}) &< Q\,t_{ora} + q_u t_{ora} \\ Q\,t_{sem} + Q\,t_{ora} &< Q\,t_{ora} + q_u t_{ora} \\ Q\,t_{sem} &< q_u t_{ora} \\ t_{sem} &< \frac{q_u}{Q} t_{ora} \end{aligned} \tag{5.6}$$

The ratio q_u/Q in Equation 5.6 may be interpreted as the *probability of an unsatisfiable query*, P_u. So, for positive optimization we must have:

$$t_{sem} < P_u t_{ora} \tag{5.7}$$

Suppose the probability of an unsatisfiable query occurring is 10%. Then, on average, t_{sem} must be a factor of ten less than t_{ora} to break even. That is, the average time required to semantically optimize a query cannot exceed one tenth of the normal time required to parse, execute and fetch the query. This places an upper bound on the amount of semantic optimization we can expect in the presence of unsatisfiable queries.

5.4.2.3 Linear relationship

Consider Equation 5.5. We may write:

$$\begin{aligned}
\frac{T_2}{T_1} &= \frac{Q\,(t_{sem} + t_{ora}) - q_u t_{ora}}{Q\;t_{ora}} \\
&= \frac{Q\,(t_{sem} + t_{ora})}{Q\;t_{ora}} - \frac{q_u t_{ora}}{Q\;t_{ora}} \\
&= -\frac{q_u}{Q} + \frac{(t_{sem} + t_{ora})}{t_{ora}} \\
\frac{T_2}{T_1} &= -P_u + \frac{(t_{sem} + t_{ora})}{t_{ora}} \qquad (5.8)
\end{aligned}$$

If we assume the times t_{sem} and t_{ora} are constant, then Equation 5.8 predicts a linear relationship between the batch time ratio T_2/T_1 and the probability of an unsatisfiable query P_u. Furthermore, Equation 5.8 predicts a line gradient of -1. Consider the case when $P_u = 0$; i.e., all queries are *positive*. Then if $t_{sem} << t_{ora}$ we expect the intercept on the y-axis to be 1. Equivalently, we can never do better than $t_{sem} = 0$; i.e., we take a negligible time to semantically optimize each query. Figure 5.4 graphs the relationship between the batch time ratio and the probability of an unsatisfiable query predicted by this cost model for $t_{sem} << t_{ora}$.

Our cost model above predicts a straightforward relationship between P_u, the probability of an unsatisfiable query, and the ratio of the optimized batch time to the normal batch time. These predictions are one of the major motivators for the formal hypotheses and experiments which are described in detail in Chapter 6 “Empirical Results”.

5.4.2.4 Cost metrics

In this Section we have developed our cost model by supposing it is the *query execution time* we are measuring. But we could equally well have measured *total CPU time* used to process the query or the *amount of disk i/o* required to process the query. In each case, an analogous argument to the above could have been developed and in each case it is the *ratio* of the optimized to unoptimized metric that we wish to measure. We therefore expect the cost model to apply to all three of these metrics.

In fact, we argue in Section 5.2.4 above that it is more meaningful to consider the *average* of the three metric ratios , which we denote by R_{com}.

5.4.3 Cost Model: Zero Queries

In Section 4.4.2 (page 109) we showed how the detection of data holes can be used to enhance the efficiency of SQO. The critical point of this section is that once data holes have been discovered in a column C_i, this information can be added to existing constraints on the allowable range of values for column C_i, effectively increasing the probability of detecting unsatisfiable queries against C_i. The cost model developed above in Section 5.4.2 is therefore applicable also to zero queries.

5.4.4 Cost Model: Unsatisfiable Joins

The cost model developed above in Section 5.4.2 could have been developed by considering unsatisfiable *joins*, as opposed to unsatisfiable queries. This is because the semantic optimizer is configured as a preprocessor and we make the assumption that the cost of preprocessing a query/join is approximately constant. In the case of joins however, we expect this preprocessing cost to be more significant. This is described in detail in Section 4.4.3.2 (page 116). The cost model developed above in Section 5.4.2 is therefore applicable also to zero queries.

5.4.5 Cost Model: "`distinct`" and "`is not null`" removal

We now develop a cost model to predict the effect of removing the phrases "`distinct`" and "`is not null`" in situations where they are redundant. Both of these are examples of restriction removal, which was described in Section 2.5.2 (page 31). In the case of "`distinct`" removal, the advantage is gained by removing the necessity to sort the result set. In the case of "`is not null`" removal, the advantage is gained by removing the necessity to check each member of the result set is non-null. We make the following simplifying assumptions:

1. The cost of sorting the result set is directly proportional to the cardinality of the result set;

2. The cost of checking each member of the result set is non-null is directly proportional to the cardinality of the result set.

We write the time for a normal unoptimized query batch as T_1 and the time for the equivalent optimized batch as T_2. T_{sort} is the total time taken to sort all result sets in the batch. T_{null} is the total time taken to check each member of the result set is non-null.

5.4.5.1 "`distinct`" removal

Suppose these two batches are identical except the optimized batch does not need to be sorted. Therefore:

$$T_2 = T_1 - T_{sort}$$

where T_{sort} is the total time taken to sort all result sets in the batch. So the ratio of the optimized to the unoptimized batch time is just:

$$\frac{T_2}{T_1} = \frac{T_1 - T_{sort}}{T_1} \tag{5.9}$$

If the cardinality of the result set is small, T_{sort} will be negligible and the ratio $\frac{T_2}{T_1}$ will approach 1. The cost model also dictates $\frac{T_2}{T_1}$ to be constant across all result set cardinalities (because of assumption 1 above in Section 5.4.5).

5.4.5.2 "`is not null`" removal

Exactly the same argument can be applied, by substituting the redundant sort time, t_{sort}, with the redundant check time (that each member of the result set is non-null), t_{chk}. In this case, the ratio of the optimized to the unoptimized batch time is:

$$\frac{T_2}{T_1} = \frac{T_1 - T_{chk}}{T_1} \tag{5.10}$$

If the cardinality of the result set is small, T_{chk} will be negligible and the ratio $\frac{T_2}{T_1}$ will approach 1. The cost model also dictates $\frac{T_2}{T_1}$ to be constant across all result set cardinalities (because of assumption 2 above in Section 5.4.5).

5.5 Summary

In this Chapter we present the methodology we employ to carry out a series of empirical investigations whose overall aim is to demonstrate the efficacy of our semantic query optimizer. The main contributions of this Chapter include the following.

- We explain the difficulty of obtaining consistent, repeatable empirical results with RDBMS where automatic maintenance processes may execute at unpredictable times and where large memory caches are available and how we remediate this problem (Section 5.2.2).

- We describe how we simulate a busy database environment by querying six non-identical tables of the same relative size, sharing the same data distribution (Section 5.2.3).

- We describe how to get the Oracle RDBMS itself to measure with great accuracy the true cost of query execution and how we use the *ratio* of the optimized over the unoptimized cost to minimize systematic error (Section 5.2.4).
- We explain why judging query cost by elapsed time alone may be misleading and why we use the average of three metric ratios to convey a more meaningful measure of query cost (Section 5.2.5).
- We describe how a query normal form arises naturally from our earlier definition of the *interval list* data type (Section 5.3.1).
- We describe a qualitative method for judging query difficulty which we subsequently use in our experiments (Section 5.3.2).
- We derive a cost model which predicts the relationship between *efficiency gain due to semantic optimization* and the *probability of an unsatisfiable query* being submitted (Section 5.4).
- We derive a cost model which predicts the *efficiency gain due to the removal of redundant "`distinct`" and "`is not null`" phrases* (Section 5.4.5).

Chapter 6

Empirical Results and Analysis

6.1 Introduction

In this chapter, we follow the description of our experimental methodology in Chapter 5 with a full review of our experimental results. We demonstrate the effectiveness of the different types of SQO we described first in Chapter 2 and expanded upon in Chapter 4.

A critical feature of our experiments is that the database schema we employ is representative of of schemas found in a wide range of real database applications. We make a minimum of assumptions as to the actual schema details, instead giving averaged results for query batches against large[1] tables with columns that are *indexed*. To our knowledge, these are the first comprehensive empirical results which demonstrate SQO for queries with restrictions and equi-joins, in the presence of standard B-tree indexes. This is important because it is most unlikely in practice that tables with similar characteristics to the ones we query in our experiments would *not* be indexed. Other writers have demonstrated the effectiveness of SQO only in limited circumstances where typically only a small number of queries have been optimized (sometimes manually), table sizes are relatively small and target columns are not indexed (Gryz et al. 1999, Cheng et al. 1999).

All results reported are derived from *batches* of similar queries. Each batch comprised between 100 and 1000 queries depending on the total time (typically between 1 and 4 hours) required for the batch to complete. For example, batches of simple queries against a single target table consisted of 1000 individual queries whereas batches of equi-joins between two target tables consisted of 100 individual queries. We do not attempt to predict outcomes for individual queries. Our results are a statistical average and our objective is to identify the precise circumstances for which semantic query optimization is likely to be worthwhile.

We refer in the following sections to the number of "restrictions per query" (R/Q) and to the number of "intervals per restriction" (I/R). We use these terms in the specific sense of *query difficulty* as we define it in Section 5.3 (page 141) where we described our qualitative classification of query difficulty. We evaluate the relative difficulty of a query by noting the number of restriction clauses (*vertical difficulty*) and the number of intervals described by each restriction (*horizontal difficulty*). Our motivation is the notion that more complex queries are likely to require more resources to answer and indeed this is precisely what our experimental results below confirm.

The experiments we report below are divided into the following three groups:

[1]We will quantify what we mean by "large" in due course but in the meantime we mean tables that are large enough to provoke significant disk activity and provoke the SQL optimizer to consult relevant indexes.

1. *Unsatisfiable queries and joins*: The objective of these experiments is to reveal the relationship between the *gain in query efficiency* and the *probability of an unsatisfiable query or join* in the presence of a semantic query preprocessor (Sections 6.3 to 6.9).

2. *Removal of redundant SQL phrases*: The objective of these experiments is to confirm or otherwise the efficacy of simplifying the SQL query text by removing certain phrases which a semantic query optimizer deduces are redundant (Sections 6.10 to 6.13).

3. *Restriction introduction and removal*: The objective of these experiments is to investigate the effect of introducing or removing restrictions into the SQL query text which a semantic query optimizer deduces will reduce the cardinality of the result set (Sections 6.14 to 6.15).

We begin each experiment by stating formal hypotheses. This is followed by a description of each experiment, a graphical summary of results and comments concerning the significance of these results. For clarity, we present only summary graphs for each experiment, which display results for the *combined* metric ratio (Definition 5.2.1, page 140) which we derived in Chapter 5. A full set of results corresponding to each experiment from the first group may be found in Appendix A.

The remainder of this chapter is organised as follows.

- In Section 6.2 we describe the format of our experimental results.

- In Section 6.3 we report baseline results for experiments with unsatisfiable queries on tables that are *not* indexed. We investigate the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size.*

- In Section 6.4 we report results for experiments with unsatisfiable queries on tables that *are* realistically indexed. We investigate the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size.*

- In Section 6.5 we report results for experiments with unsatisfiable queries on indexed tables where we investigate the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *number of restrictions per query.*

- In Section 6.6 we report results for experiments with unsatisfiable queries on indexed tables where we investigate the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *number of intervals per restriction.*

- In Section 6.7 we report results for experiments with unsatisfiable joins on indexed tables where we investigate the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and *relative table size.*

- In Section 6.8 we report results for experiments with unsatisfiable joins on indexed tables where we investigate the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and *number of restrictions per join.*

- In Section 6.9 we report results for experiments with unsatisfiable joins on indexed tables where we investigate the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and *number of intervals per restriction.*

- In Section 6.10 we investigate the efficacy of eliminating the key word "`distinct`" in queries in which it is redundant in the context of the select clause "`select distinct`".

- In Section 6.11 we investigate the efficacy of eliminating the key word "`distinct`" in joins in which it is redundant in the context of the select clause "`select distinct`".

- In Section 6.12 we investigate the efficacy of eliminating the key phrase "`is not null`" in queries in which it is redundant in the context of the *where* clause "`where COL is not null`".

- In Section 6.13 we investigate the efficacy of eliminating the key phrase "`is not null`" in joins in which it is redundant in the context of the *where* clause "`where COL is not null`".

- In Section 6.14 we investigate the efficacy of *restriction introduction and removal* with queries.

- In Section 6.15 we investigate the efficacy of *restriction introduction and removal* with joins.

- Finally in Section 6.16 we summarise the main contributions of this chapter.

6.2 Format of Experimental Results

We use a three dimensional graphical projection to present important result summaries. In each case:

- the X-axis is the independent variable *probability of an unsatisfiable query* in a given batch of queries;
- the Y-axis is another independent variable, one of *number of table rows*, *number of restrictions per query* or *number of intervals per restriction*;
- the Z-axis is the dependent variable *cost metric ratio*.

The "ruggedness" of this surface corresponds to the variation or uncertainty in the actual recorded data. An example is displayed in Figure 6.1. This is plotted as a *red* surface. This category of graph may include the following features.

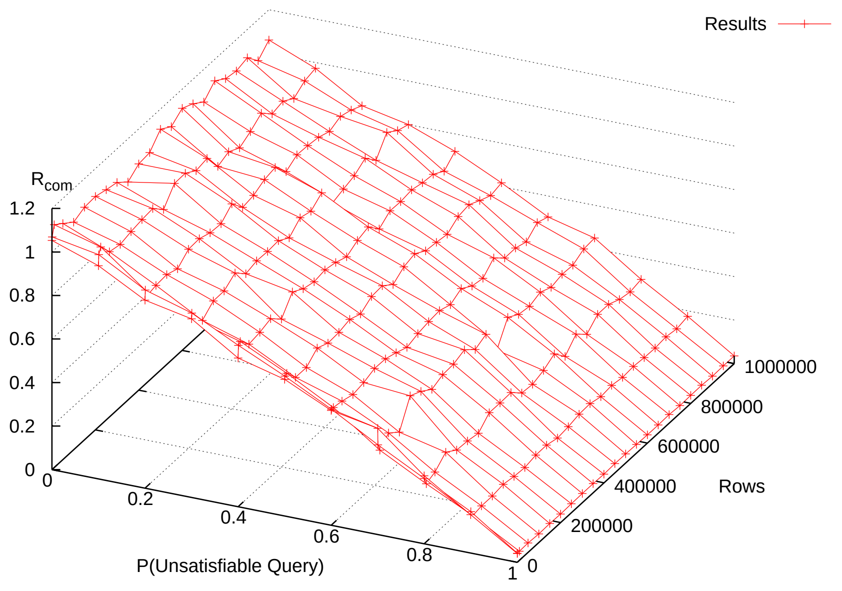


Figure 6.1: **Example 3D projection of experimental results**: The X-axis is the independent variable *probability of an unsatisfiable query*. The Y-axis is another independent variable such as *number of table rows*. The Z-axis is the dependent variable and depicts the *cost metric ratio*, for example, R_{com}. The "ruggedness" of this surface corresponds to the variation or uncertainty in the actual recorded data. Experimental results are always plotted in *red*.

- *Cost Model Surface*: This is the plot of $f(x) = 1-x$ and represents the surface predicted by the cost model developed in Section 5.4 (page 144) *where the time taken by the extra semantic optimizing step is negligible*. Therefore, in the absence of any other optimization, we cannot reasonably expect the cost metric ratio to be below this surface. This is plotted as a *blue* surface. See Figure 6.2.
- *Break Even Surface*: This surface marks the cost metric ratio of 1, representing equal costs for both optimized and normal batches. Therefore, any

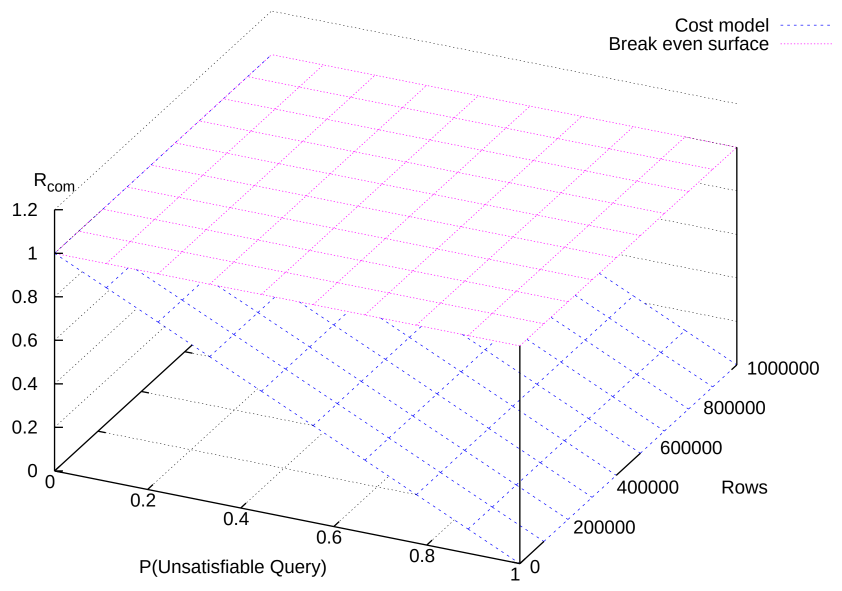


Figure 6.2: **Cost Model Surface and Break Even Surface**: Results that conform closely to the idealised cost model will appear near to and parallel to the "cost model" surface. Result surfaces that appear below the "break even" surface indicate a positive optimization. The cost model surface is always plotted in *blue*. The break even surface is always plotted in *pink*.

results below this surface represent a *positive* optimization; i.e., the semantically optimized cost is less than the normal cost. Conversely, any results above this surface represent a *negative* optimization; i.e., the semantically optimized cost is actually more than the normal cost. This is plotted as a *pink* surface. See Figure 6.2.

- *Regression surface*: We calculate a *regression surface* for each result set using an implementation of the nonlinear least-squares (NLLS) *Marquardt-Levenberg algorithm* (Press, Flannery, Teukolsky & Vetterling 1992) as implemented by *Gnuplot* (Broeker, Campbell, Cunningham & Denholm 2006, Drakos & Moore 2006)[2]. The form of the regression surface is given by the following:

$$f(x,y) = A + Bx + Cx^2 + Dy + Ey^2$$

where $f(x,y)$ is the dependent variable, x and y are the independent variables, A,B,C,D,E are constants determined by the regression analysis. This is plotted as a *green* surface. See Figure 6.3.

[2]*Gnuplot* is a portable command-line driven interactive data and function plotting utility. See `http://www.gnuplot.info`.

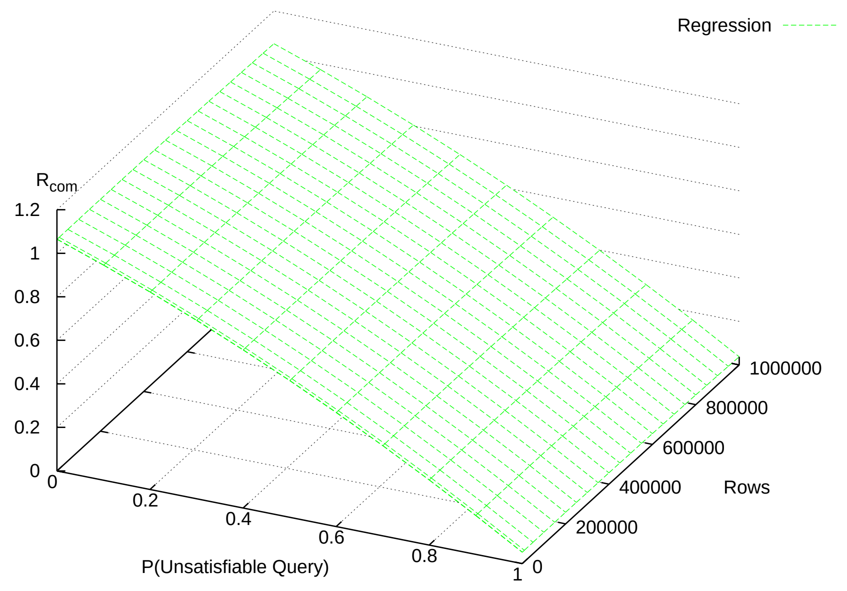


Figure 6.3: **Example regression surface**: The regression surface is calculated using an implementation of the nonlinear least-squares (NLLS) *Marquardt-Levenberg algorithm* and displays the relationship between the the dependent variable (cost metric ratio on the Z-axis) and the two independent variables. The regression surface is always plotted in *green*.

6.3 Unsatisfiable Queries – No Indexing

The objective of these experiments is to establish a baseline with regard to the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size*. It is most unlikely in practice that tables of the size and makeup we query in our experiments would *not* be indexed. However, we are motivated to query a set of unindexed tables:

- to set a baseline against which our other experiments with tables that *are* realistically indexed may be compared;
- to relate our work with other published research that typically cite results for unindexed tables (Gryz et al. 1999, Cheng et al. 1999).

6.3.1 Hypotheses

1. The gain in query efficiency increases linearly with increasing probability of an unsatisfiable query and will be close to the idealised cost model of Section 5.4.2.3 (page 148).
2. The gain in query efficiency is independent of table size.

6.3.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query *P*
- Relative table size, denoted by the number of table rows *Rows*.

The dependent variable is the combined cost metric ratio R_{com} as defined in Table 5.1 (page 139). Each query consists of a single restriction defined by one interval. None of the columns cited in query restrictions is indexed.

Example 6.3.1. *The following is a typical query drawn from the batch used in these experiments:*

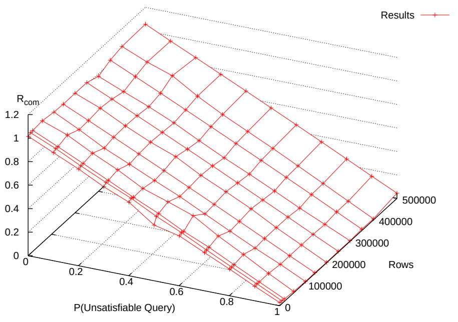
```
select t.COL3, t.COL7, t.COL9, t.COL5, t.COL2
from   TAB3 t
where  t.COL1 >= 47682608
and    t.COL1 < 47682656;
```

6.3.3 Results and Analysis

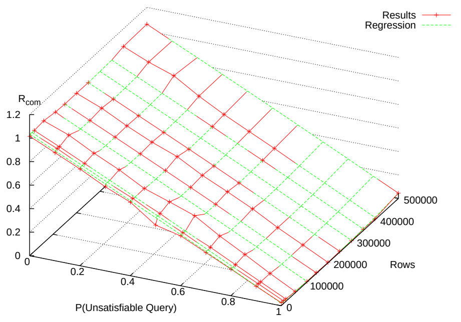
We present summary results which show the relationship between *P*, *Rows* and R_{com} in Figure 6.4. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables *P* and *Rows* (Figure 6.4(a))
- The cost metric ratio surface R_{com} with the regression surface superimposed (Figure 6.4(b))
- The regression surface with “cost model” and “break even” surfaces (Figure 6.4(c))
- The cost metric ratio surface R_{com}, regression surface, “cost model” and “break even” surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size *Rows*) disappears (Figure 6.4(d))

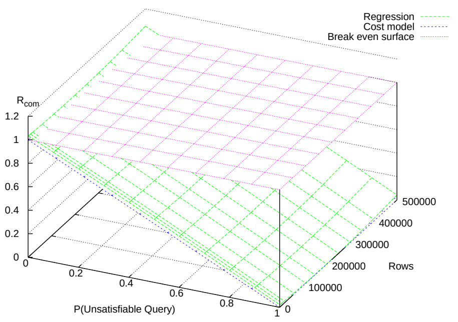
Figures 6.4(b) and 6.4(c) show R_{com} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size, with the optimization cost rising slightly as table size becomes very large ($Rows > 400,000$).



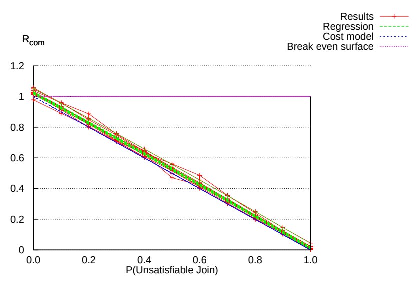
(a) R_{com} surface for `100` to `500,000` rows



(b) R_{com} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure 6.4: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (no indexing)**: Figures 6.4(b) and 6.4(c) show R_{com} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size, with the optimization cost rising slightly as table size becomes very large ($Rows > 400,000$).

6.3.4 Conclusion

The cost model of Section 5.4.2.3 (page 148) accurately predicts the dependence of *gain in query efficiency* on *probability of an unsatisfiable query*. Crucially, this cost model sets an upper bound for the amount of optimization we can expect from detecting unsatisfiable queries. That is, even if the cost of detecting unsatisfiable queries is negligible (and it is not), the maximum amount of optimization is irrevocably determined by the prevalence of unsatisfiable queries. The hypotheses are confirmed.

6.4 Indexed Unsatisfiable Queries

The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size*. The methodology is identical to the experiments reported above in Section 6.3, with the key difference that all columns cited in query restrictions are indexed with a "normal" B-tree index (Chan 2005*a*).

6.4.1 Hypotheses

1. The gain in query efficiency increases linearly with increasing probability of an unsatisfiable query and will be close to the idealised cost model of Section 5.4.2.3 (page 148).

2. The gain in query efficiency is independent of table size.

6.4.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P

- Relative table size, denoted by the number of table rows $Rows$.

The dependent variable is the combined cost metric ratio R_{com} as defined in Table 5.1 (page 139). Each query consists of a single restriction defined by one interval.

Example 6.4.1. *The following is a typical query drawn from the batch used in these experiments:*

```
select t.COL4, t.COL2, t.COL8, t.COL10
from   TAB6 t
where  t.COL5 >= 52310468
and    t.COL5 < 52310572;
```

6.4.3 Results and Analysis

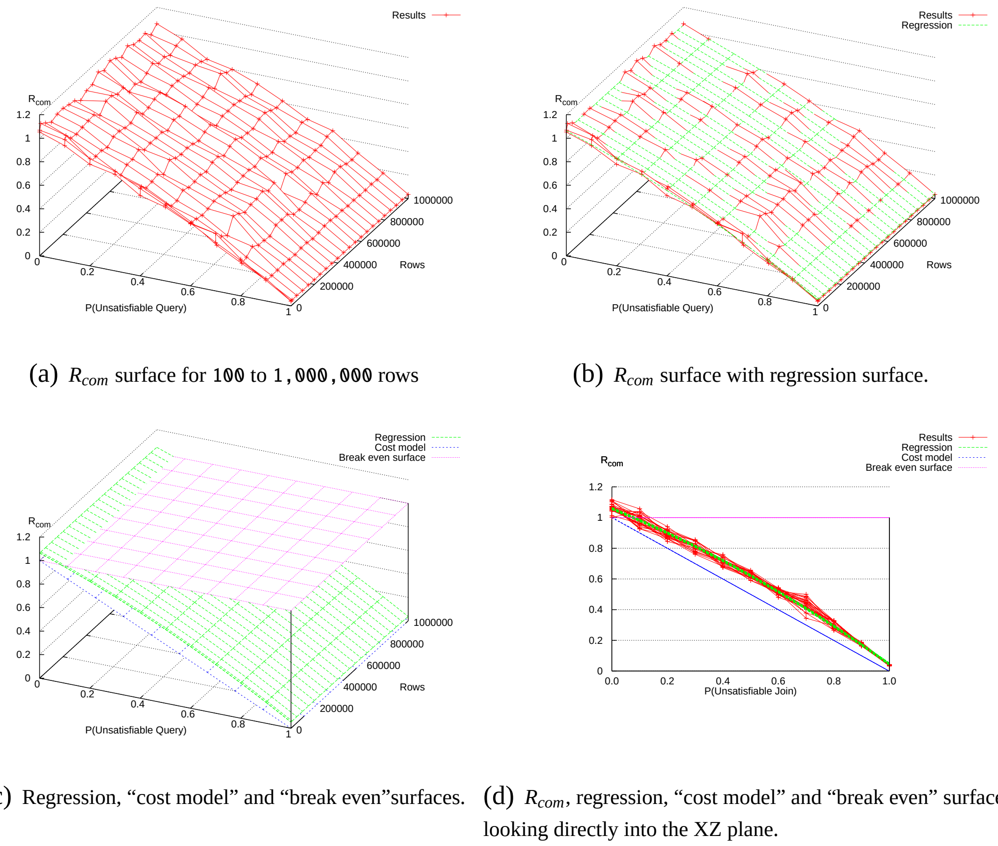
We present summary results which show the relationship between P, $Rows$ and R_{com} in Figure 6.5. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and $Rows$

- The cost metric ratio surface R_{com} with the regression surface superimposed

- The regression surface with "cost model" and "break even" surfaces

- The cost metric ratio surface R_{com}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size $Rows$) disappears



(a) R_{com} surface for `100` to `1,000,000` rows

(b) R_{com} surface with regression surface.

(c) Regression, "cost model" and "break even"surfaces.

(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure 6.5: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (indexed)**: Figures 6.5(b) and 6.5(c) show the R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.1$.

Figures 6.5(b) and 6.5(c) show the R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.1$.

6.4.4 Conclusion

This series of experimental results confirms that the detection of unsatisfiable queries can significantly enhance overall query efficiency by preventing such queries being submitted to the database. Our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of this efficiency gain on the probability of unsatisfiable queries occurring. This efficiency gain is maintained across at least four orders of magnitude of table size and we conclude the effect is so strong as to be almost independent of table size. Crucially, our target tables are all sensibly indexed so our results are realistic and we have a high expectation that such results will be confirmed in commercial database environments.

Comparison of these results with those presented above in Section 6.3 reveals that in the case of unindexed tables, the cost of detecting unsatisfiable queries is negligible. However, when table columns are sensibly indexed, this cost cannot be discounted.

We therefore conclude semantic query optimization is worthwhile in the presence of unsatisfiable queries, provided the probability of such queries being submitted is not vanishingly small. For our prototype reasoning engine, a threshold probability of approximately 10% justifies the extra processing required by the semantic optmizer. The hypotheses are confirmed.

6.5 Indexed Unsatisfiable Queries – Varying Restrictions per Query

The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and the *number of restrictions per query*. The methodology is identical to the experiments reported above in Section 6.4, except that we hold table size constant at $1,000,000$ rows while varying number of restrictions per query R/Q.

6.5.1 Hypotheses

1. The gain in query efficiency increases linearly with increasing probability of an unsatisfiable query and will be close to the idealised cost model of Section 5.4.2.3 (page 148).
2. The gain in query efficiency is degraded by increasing query complexity.

6.5.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Number of restrictions per query R/Q. Each restriction is defined by a single interval.

The dependent variable is the combined cost metric ratio R_{com} as defined in Table 5.1 (page 139). All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in query restrictions are indexed with a "normal" B-tree index.

Example 6.5.1. *The following is a typical query drawn from the batch used in these experiments. In this example, $R/Q = 3$:*

```
select t.COL10
from   TAB2 t
where  (t.COL5 > 52042096 and t.COL5 < 52042200)  -- Restriction 1
and    (t.COL3 > 50721696 and t.COL3 <= 50721744) -- Restriction 2
and    (t.COL4 > 50841160 and t.COL4 <= 50841264) -- Restriction 3
```

6.5.3 Results and Analysis

We present summary results which show the relationship between P, R/Q and R_{com} in Figure 6.6. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and R/Q
- The cost metric ratio surface R_{com} with the regression surface superimposed
- The regression surface with “cost model” and “break even” surfaces
- The cost metric ratio surface R_{com}, regression surface, “cost model” and “break even” surfaces viewed by looking directly into the XZ plane such that the Y axis (restrictions per query R/Q) disappears

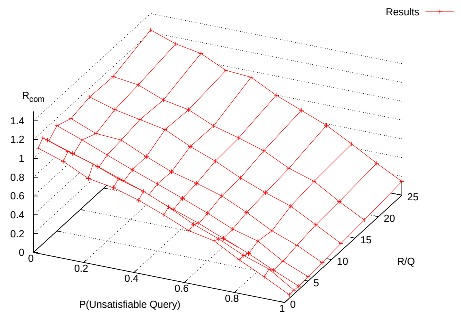
As the number of Restrictions per Query R/Q increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $P = 10\%$, positive optimization is achieved when there is up to five restrictions per query; i.e., $R/Q \leq 5$.

6.5.4 Conclusion

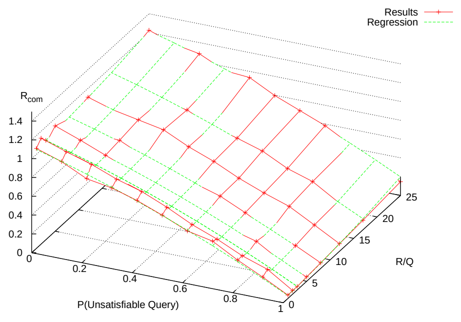
For low values of R/Q, our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of gain in query efficiency with probability of an unsatisfiable query, yielding results identical to those presented above in Section 6.4. However, as the number of restrictions per query rises, the increasing query complexity requires the semantic optimizer to do more work. A greater and greater proportion of unsatisfiable queries is required to “break even”. The hypotheses are confirmed.

6.6 Indexed Unsatisfiable Queries – Varying Intervals per Restriction

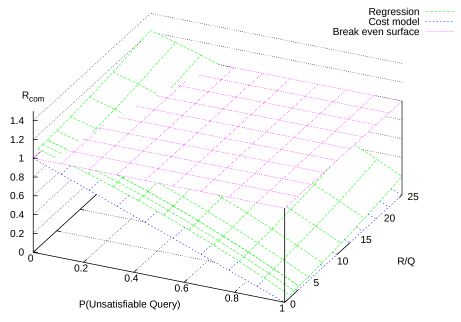
The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and the *number of intervals per restriction*. The methodology is identical to the experiments reported



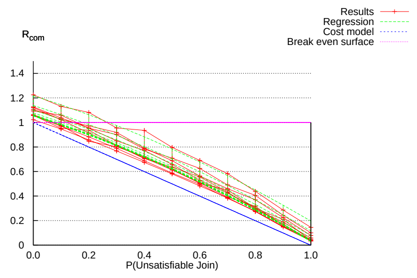
(a) R_{com} surface for $R/Q = 1$ to 25.



(b) R_{com} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure 6.6: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: As the number of Restrictions per Query R/Q increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $P = 10\%$, positive optimization is achieved when there is up to five restrictions per query; i.e., $R/Q \leq 5$. Number of table rows $Rows = 1,000,000$.

above in Section 6.4, except that we hold table size constant at $1,000,000$ rows while varying number of intervals per restriction I/R.

6.6.1 Hypotheses

1. The gain in query efficiency increases linearly with increasing probability of an unsatisfiable query and will be close to the idealised cost model of Section 5.4.2.3 (page 148).
2. The gain in query efficiency is degraded by increasing query complexity.

6.6.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Number of intervals per restriction I/R. Each query has a single restriction.

The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139). All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in query restrictions are indexed with a “normal” B-tree index.

Example 6.6.1. *The following is a typical query drawn from the batch used in these experiments. In this example, $I/R = 3$:*

```
select t.COL7, t.COL4, t.COL3
from   TAB2 t
where (
  (t.COL5 > 52042096 and t.COL5 < 52042200)  or -- Interval 1
  (t.COL5 >= 52287616 and t.COL5 < 52287668) or -- Interval 2
  (t.COL5 >= 52310468 and t.COL5 < 52310572)    -- Interval 3
)
```

6.6.3 Results and Analysis

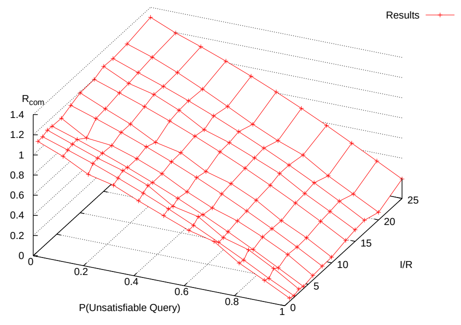
We present summary results which show the relationship between P, I/R and R_{com} in Figure 6.7. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and I/R
- The cost metric ratio surface R_{com} with the regression surface superimposed
- The regression surface with “cost model” and “break even” surfaces
- The cost metric ratio surface R_{com}, regression surface, “cost model” and “break even” surfaces viewed by looking directly into the XZ plane such that the Y axis (restrictions per query I/R) disappears

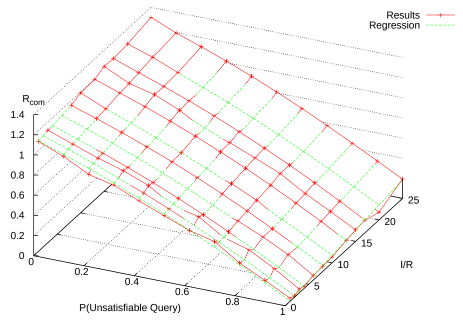
As the number of Intervals per Restriction I/R increases from 1 to 25, ratio R_{com} increases slowly. For $P > 0.15$, positive optimization is achieved throughout the whole range.

6.6.4 Conclusion

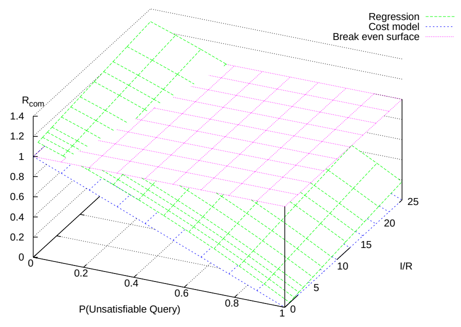
For low values of I/R, our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of gain in query efficiency with probability of an unsatisfiable query, yielding results identical to those presented above in Section 6.4. However, as the number of intervals per restriction rises, the increasing query complexity requires the semantic optimizer to do more work. A greater and greater proportion of unsatisfiable queries is required to “break even”. The hypotheses are confirmed.



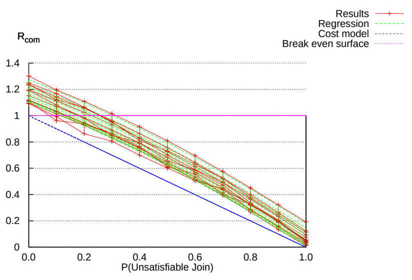
(a) R_{com} surface for I/R = 1 to 25



(b) R_{com} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure 6.7: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: As the number of Intervals per Restriction I/R increases from 1 to 25, ratio R_{com} increases slowly. For $P > 0.15$, positive optimization is achieved throughout the whole range. Number of table rows $Rows = 1,000,000$.

6.7 Indexed Unsatisfiable Joins

The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and *relative table size*. The methodology is identical to the experiments reported above in Section 6.4, except that we submit batches of equi-joins between two tables rather than simple queries against a single table.

6.7.1 Hypotheses

1. The gain in join efficiency increases linearly with increasing probability of an unsatisfiable join and will be close to the idealised cost model of Section 5.4.2.3 (page 148).

2. The gain in join efficiency is independent of table size.

6.7.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P
- Relative table size, denoted by the number of table rows *Rows*.

The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139). Each join consists of a single join clause citing the equi-join columns plus a single restriction defined by one interval.

Example 6.7.1. *The following is a typical join drawn from the batch used in these experiments:*

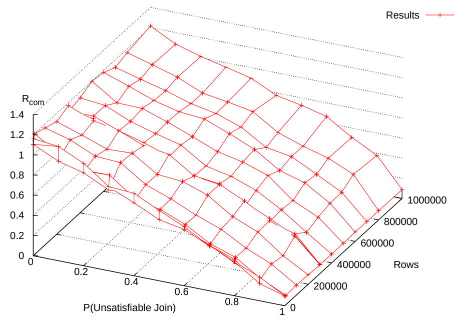
```
select t1.COL6 c1, t2.COL10 c2
from   TAB2 t1, TAB6 t2
where  t2.COL1 = t1.COL3
and    t2.COL1 > 47128904
and    t2.COL1 < 47223256;
```

6.7.3 Results and Analysis

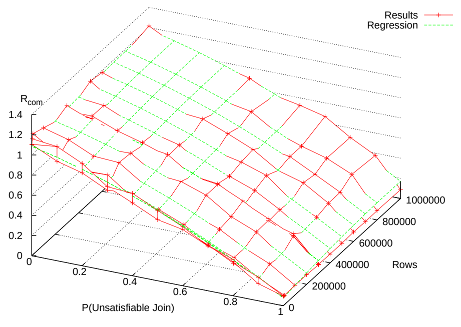
We present summary results which show the relationship between P, *Rows* and R_{com} in Figure 6.8. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and *Rows*
- The cost metric ratio surface R_{com} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{com}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size *Rows*) disappears

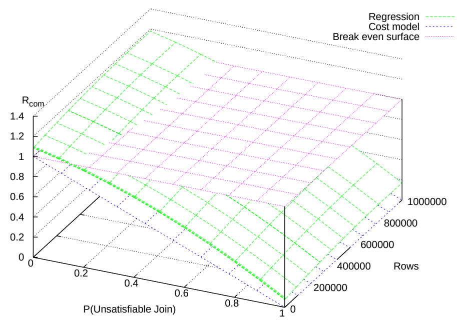
Figures 6.8(b) and 6.8(c) show R_{com} surface sits just above the "cost model surface", indicating results deviate very little from those predicted by the cost model. We have positive optimization across four orders of magnitude of table size when $P > 0.2$, although it is evident that the cost of processing the semantically optimized joins increases relatively as table size becomes very large.



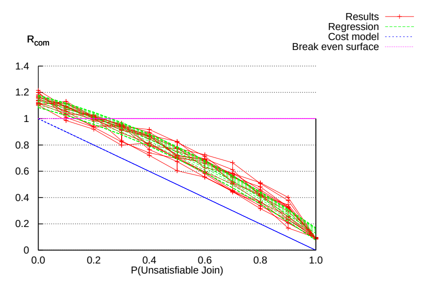
(a) R_{com} surface for `100` to `1,000,000` rows



(b) R_{com} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure 6.8: **Ratio R_{com} vs Probability of Unsatisfiable Join P vs Relative Table Size $Rows$ (indexed)**: Figures 6.8(b) and 6.8(c) show R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.2$, although it is evident that the cost of processing the semantically optimized joins increases relatively as table size becomes very large.

6.7.4 Conclusion

The results we obtain for semantically optimized joins are analogous to the results reported above in Section 6.4 for queries. The detection of unsatisfiable joins can significantly enhance overall join efficiency by preventing such joins being submitted to the database. Our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of this efficiency gain on the probability of unsatisfiable joins occurring. This efficiency gain is maintained across at least four orders of magnitude of table size and we conclude the effect is strong but not entirely independent of table size. Crucially, our target tables are all sensibly indexed so our results are realistic and we have a high expectation that such results will be confirmed in commercial database environments.

Comparison of these results with those presented above in Section 6.4 reveals that in the case of queries, the cost of detecting unsatisfiable queries is relatively smaller than the cost of detecting unsatisfiable joins.

We therefore conclude semantic query optimization is worthwhile in the presence of unsatisfiable joins, provided the probability of such joins being submitted is high enough. For our prototype reasoning engine, a threshold probability of approximately 20% justifies the extra processing required by the semantic optmizer. The hypotheses are confirmed, but with regard to the caveats expressed above.

6.8 Indexed Unsatisfiable Joins – Varying Restrictions per Join

The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and the *number of restrictions per join*. The methodology is identical to the experiments reported above in Section 6.5, except that we submit batches of equi-joins between two tables rather than simple queries against a single table.

6.8.1 Hypotheses

1. The gain in join efficiency increases linearly with increasing probability of an unsatisfiable join and will be close to the idealised cost model of Section 5.4.2.3 (page 148).
2. The gain in join efficiency is degraded by increasing join complexity.

6.8.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P
- Number of restrictions per join R/Q. Each restriction is defined by a single interval.

The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139). All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in join restrictions are indexed with a “normal” B-tree index.

Example 6.8.1. *The following is a typical join drawn from the batch used in these experiments. In this example, $R/Q = 3$:*

```
select t1.COL10, t2.COL1
from   TAB2 t1, TAB5 t2
where  t1.COL3 = t2.COL4
```

```
and    (t1.COL5 >= 52042096 and t1.COL5 < 52042200) -- Restriction 1
and    (t2.COL3 > 50721696 and t2.COL3 <= 50721744) -- Restriction 2
and    (t1.COL4 > 50841160 and t1.COL4 <= 50841264) -- Restriction 3
```

6.8.3 Results and Analysis

We present summary results which show the relationship between P, R/Q and R_{com} in Figure 6.9. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and R/Q
- The cost metric ratio surface R_{com} with the regression surface superimposed
- The regression surface with “cost model” and “break even” surfaces
- The cost metric ratio surface R_{com}, regression surface, “cost model” and “break even” surfaces viewed by looking directly into the XZ plane such that the Y axis (restrictions per query R/Q) disappears

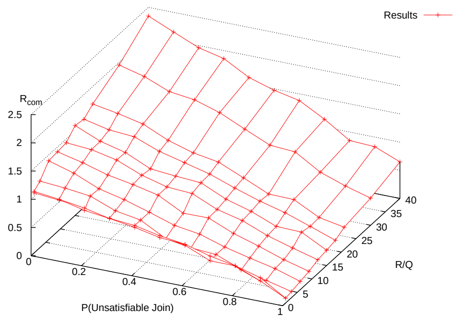
Figure 6.9 plots R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q and summarises the results as a R_{com} surface. For low R/Q, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the “cost model surface”. However as R/Q rises, the pre-processing cost becomes significant and we require a greater proportion of unsatisfiable queries to make semantic optimization worthwhile.

6.8.4 Conclusion

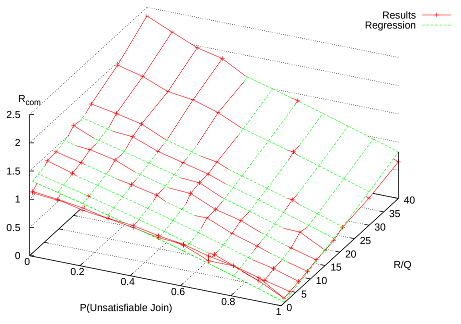
For low values of R/Q, our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of gain in join efficiency with probability of an unsatisfiable join, yielding results very similar to those presented above in Section 6.7. However, as the number of restrictions per join rises, the increasing join complexity requires the semantic optimizer to do more work. A greater and greater proportion of unsatisfiable joins is required to “break even”. The hypotheses are confirmed.

6.9 Indexed Unsatisfiable Joins – Varying Intervals per Restriction

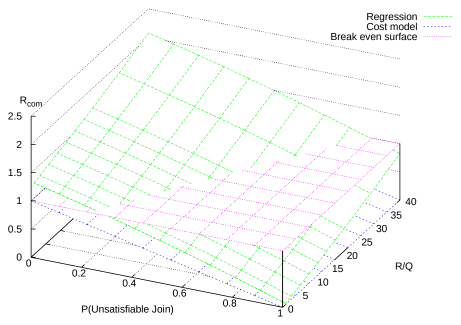
The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and the *number of intervals*



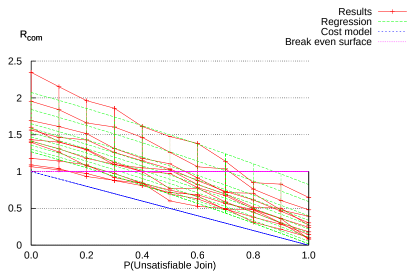
(a) R_{com} surface for $R/Q = 1$ to 40.



(b) R_{com} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure 6.9: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: As the number of Restrictions per Query R/Q increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. For $P = 0.25$, positive optimization is achieved when there is up to five restrictions per join; i.e., $R/Q \leq 5$. Number of table rows $Rows = 1,000,000$.

per restriction. The methodology is identical to the experiments reported above in Section 6.6, except that we submit batches of equi-joins between two tables rather than simple queries against a single table.

6.9.1 Hypotheses

1. The gain in join efficiency increases linearly with increasing probability of an unsatisfiable join and will be close to the idealised cost model of Section 5.4.2.3 (page 148).
2. The gain in join efficiency is degraded by increasing join complexity.

6.9.2 Method

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P

- Number of intervals per restriction I/R. Each join has a single restriction.

The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139). All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in join restrictions are indexed with a "normal" B-tree index.

Example 6.9.1. *The following is a typical query drawn from the batch used in these experiments. In this example, $I/R = 3$:*

```
select t1.COL7, t2.COL4, t2.COL3
from   TAB2 t1, TAB6 t2
where  t1.COL1 = t2.COL2
and (
  (t1.COL5 > 52042096 and t1.COL5 < 52042200)  or -- Interval 1
  (t1.COL5 >= 52287616 and t1.COL5 < 52287668) or -- Interval 2
  (t1.COL5 >= 52310468 and t1.COL5 < 52310572)    -- Interval 3
);
```

6.9.3 Results and Analysis

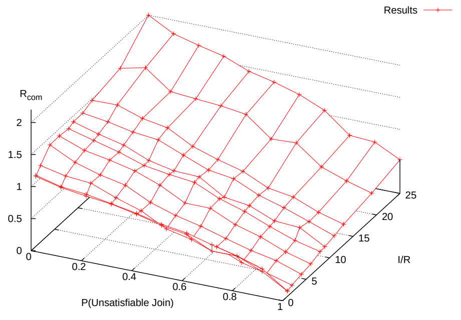
We present summary results which show the relationship between P, I/R and R_{com} in Figure 6.10. The four sub-figures depict:

- The cost metric ratio surface R_{com} plotted against the two independent variables P and I/R
- The cost metric ratio surface R_{com} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{com}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (intervals per restriction I/R) disappears

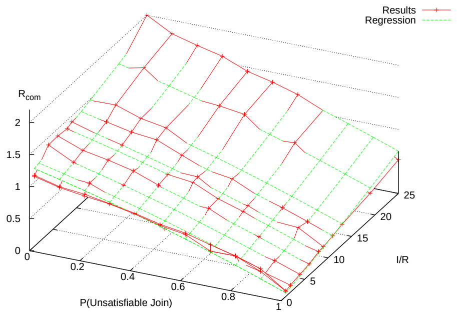
Figure 6.10 plots R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R and summarises the results as a R_{com} surface. For low I/R, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the "cost model surface". As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.25$ to achieve positive optimization.

6.9.4 Conclusion

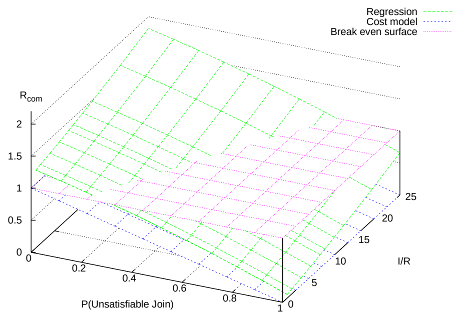
For low values of I/R, our cost model from Section 5.4.2.3 (page 148) successfully predicts the dependence of gain in join efficiency with probability of an unsatisfiable join, yielding results similar to those presented above in Section 6.7. However,



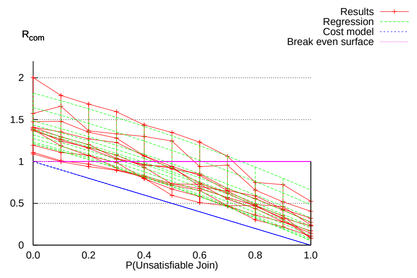
(a) R_{com} surface for I/R = 1 to 25



(b) R_{com} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure 6.10: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.25$ to achieve positive optimization. Number of table rows $Rows = 1,000,000$.

as the number of intervals per restriction rises, the increasing join complexity requires the semantic optimizer to do more work. A greater and greater proportion of unsatisfiable joins is required to "break even". The hypotheses are confirmed.

6.10 Queries "select distinct" elimination

The objective of this experiment is to confirm or otherwise the efficacy of eliminating the key word "`distinct`" in queries in which it is redundant in the context of the select clause "`select distinct`". This is described in detail in Section 4.4.3.3 where we gave a typical example (Example 4.4.7, page 118) which suggested that removal of this redundancy could halve the query cost.

6.10.1 Hypotheses

- Removing the redundant keyword "`distinct`" from the select clause "`select distinct`" will increase query efficiency.

- The effect will be negligible below a certain threshold proportional to the number of rows returned by the query.

6.10.2 Method

In this experiment we submit batches of queries to the database, identical except that one batch has all *select* clauses containing the redundant “`distinct`” keyword while the other does not. As for the previous experiments, we measure the *ratio* of the optimized batch cost (redundant “`distinct`” keyword eliminated) to the unoptimized batch cost (includes redundant “`distinct`” keyword). We reason that since the “`distinct`” keyword triggers a redundant sort, then the advantage of eliminating it will be negligible when the time required to sort the rows returned by the query is very small, relative to the total query processing time. For this reason:

- The independent variable in this case is the average number of rows returned per query: $Rows/Query$.
- The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139).

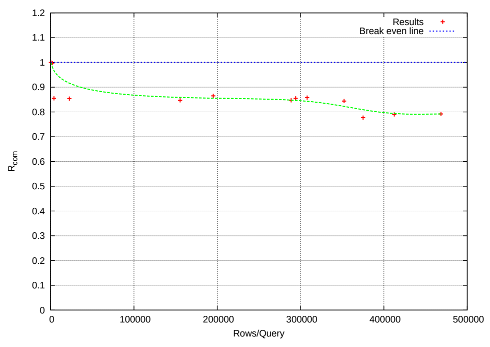
All results are for tables with number of rows $Rows = 500,000$. All queries consist of a single restriction defined by a single interval. All columns cited in query restrictions are indexed with a “normal” B-tree index.

Example 6.10.1. *The following is a typical query drawn from the batch used in these experiments:*

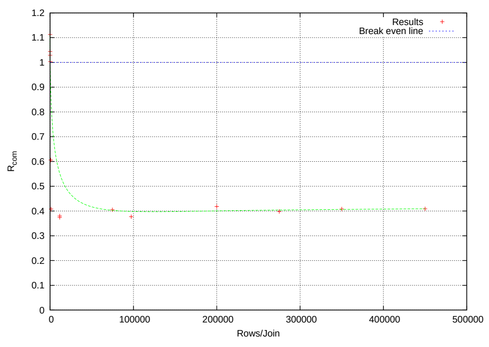
```
select distinct t.ID, t.COL7, t.COL4, t.COL3
from   TAB2 t
where  t.COL2 > 52042096
and    t.COL2 < 52042200;
```

6.10.3 Results and Analysis

Figure 6.11(a) plots R_{com} versus $Rows/Query$. This optimization works because an unnecessary sort of returned rows is avoided when the redundant keyword “`distinct`” is eliminated from the select clause of the query. When the number of rows returned is small, the sort requires negligible time so no advantage is gained. When $Rows/Query > 1,000$ the cost of the sort becomes significant and produces a nearly uniform average saving of approximately 15% to 20%. This is not quite the saving suggested by Example 4.4.7 (page 118) where the SQL optimizer predicted a saving of around 50%. However, the 50% figure is an *estimate* for just one query. The 20% result we report here is empirical for batches comprising 100 queries, so it



(a) Queries “select distinct” elimination



(b) Joins “select distinct” elimination

Figure 6.11: **“select distinct” elimination**: Eliminating the redundant “`distinct`” keyword from the *select* clause of the batch produces an average saving of approximately 15% to 20% for queries and approximately 60% for joins, as soon as the number of rows returned by the query becomes significant. This optimization works because an unnecessary sort of returned rows is avoided.

is a statistical average. Although it is beyond the scope of this experiment, we conjecture the sort cost increases suddenly in the step-wise fashion of Figure 6.11(a) because it triggers disk i/o.

6.10.4 Conclusion

Elimination of the redundant keyword “`distinct`” is worthwhile whenever the number of rows returned by the query exceeds a certain threshold. The threshold is determined by the cost of the redundant sort. The hypotheses are confirmed.

6.11 Joins “select distinct” elimination

The objective of this experiment is to confirm or otherwise the efficacy of eliminating the key word “`distinct`” in joins in which it is redundant in the context of the select clause “`select distinct`”. This is described in detail in Section 4.4.3.3 where we gave a typical example (Example 4.4.8, page 119) which suggested that removal of this redundancy could reduce the join cost by 80%.

6.11.1 Hypotheses

- Removing the redundant keyword “`distinct`” from the select clause “`select distinct`” will increase join efficiency.
- The effect will be negligible below a certain threshold determined by the number of rows returned by the join.

6.11.2 Method

In this experiment we submit batches of joins to the database, identical except that one batch has all *select* clauses containing the redundant "`distinct`" keyword while the other does not. As for the previous experiments, we measure the *ratio* of the optimized batch cost (redundant "`distinct`" keyword eliminated) to the unoptimized batch cost (includes redundant "`distinct`" keyword). We reason that since the "`distinct`" keyword triggers a redundant sort, then the advantage of eliminating it will be negligible when the time required to sort the rows returned by the join is very small, relative to the total join processing time. For this reason:

- The independent variable in this case is the average number of rows returned per join: $Rows/Join$.
- The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139).

All results are for tables with number of rows $Rows = 500,000$. All joins consist of a single restriction defined by a single interval. All columns cited in query restrictions are indexed with a "normal" B-tree index.

Example 6.11.1. *The following is a typical join drawn from the batch used in these experiments:*

```
select distinct t1.ID id1, t2.ID id2, t1.COL8 c1
from   TAB6 t1, TAB1 t2
where  t2.COL2 = t1.COL2
and    t2.COL5 > 51547332
and    t2.COL5 < 52588692;
```

6.11.3 Results and Analysis

Figure 6.11(b) plots R_{com} versus $Rows/Join$. This optimization works because an unnecessary sort of returned rows is avoided when the redundant keyword "`distinct`" is eliminated from the select clause of the join. When the number of rows returned is small, the sort requires negligible time so no advantage is gained. When $Rows/Join > 1,000$ the cost of the sort becomes significant and produces a nearly uniform average saving of approximately 60%. This is not quite the saving suggested by Example 4.4.8 (page 119) where the SQL optimizer predicted a saving of around 80%. However, the 80% figure is an *estimate* for just one query. The 60% result we report here is empirical for batches comprising 100 joins, so it is a statistical average. Although it is beyond the scope of this experiment, we conjecture the sort cost increases suddenly in the step-wise fashion of Figure 6.11(b) because it triggers disk i/o.

6.11.4 Conclusion

Elimination of the redundant keyword "`distinct`" is worthwhile whenever the number of rows returned by the join exceeds a certain threshold. The threshold is determined by the cost of the redundant sort. The hypotheses are confirmed.

6.12 Queries "is not null" elimination

The objective of this experiment is to confirm or otherwise the efficacy of eliminating the key phrase "`is not null`" in queries in which it is redundant in the context of the *where* clause "`where COL is not null`". This is described in detail in Section 4.4.3.4 where we gave a typical example (Example 4.4.9, page 120) which suggested that removal of this redundancy could halve the query cost.

6.12.1 Hypotheses

- Removing the redundant key phrase "`is not null`" from the where clause "`where COL is not null`" will increase query efficiency.

- The effect will be negligible below a certain threshold proportional to the number of rows returned by the query.

6.12.2 Method

In this experiment we submit batches of queries to the database, identical except that one batch has all *where* clauses containing the redundant "`is not null`" phrase while the other does not. As for the previous experiments, we measure the *ratio* of the optimized batch cost (redundant "`is not null`" eliminated) to the unoptimized batch cost (includes redundant "`is not null`" keyword). We reason the advantage of eliminating the phrase will be negligible when the time required to check the rows returned by the query is very small, relative to the total query processing time. For this reason:

- The independent variable in this case is the average number of rows returned per query: $Rows/Query$.

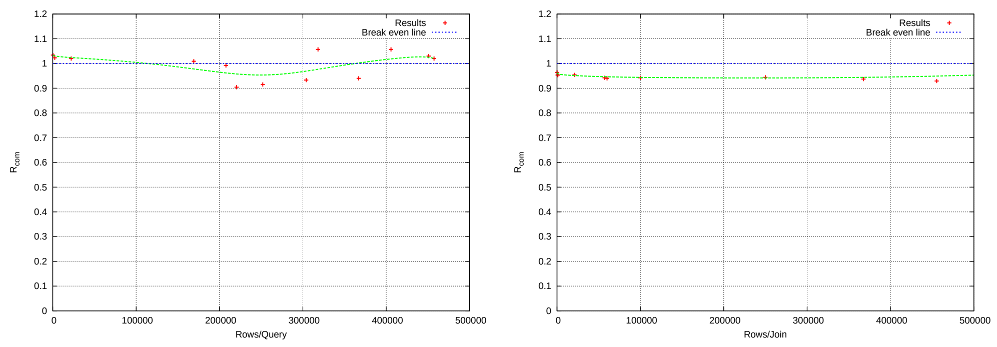
- The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139).

All results are for tables with number of rows $Rows = 500,000$. All queries consist of a single restriction defined by a single interval. All columns cited in query restrictions are indexed with a "normal" B-tree index.

Example 6.12.1. *The following is a typical query drawn from the batch used in these experiments:*

```
select distinct t.ID, t.COL7, t.COL4, t.COL3
from   TAB2 t
where  t.COL2 > 52042096
and    t.COL2 < 52042200
and    t.ID is not null;
```

6.12.3 Results and Analysis



(a) Queries "is not null" elimination (b) Joins "is not null" elimination

Figure 6.12: **"is not null" elimination**: In the case of queries, eliminating the redundant phrase from the *where* clause of the batch of queries scarcely affects query efficiency (Figure 6.12(a)). In the case of joins, the simplification produces a very small, nearly uniform saving of approximately 5% (Figure 6.12(b)).

Figure 6.12(a) plots R_{com} versus *Rows/Query*. We first considered this optimization in Example 4.4.9 (page 120) where the SQL optimizer predicted a saving of around 50%. However, our experiments with batches of queries consistently failed to show any significant optimization across a wide range of *Rows/Query*.

6.12.4 Conclusion

The results displayed in Figure 6.12(a) suggest we have gained very little or nothing by eliminating the redundant phrase "`is not null`" from the *where* clause. We conclude that, on average, for queries the effect is too small to be significant. The hypotheses are not confirmed.

6.13 Joins "is not null" elimination

The objective of this experiment is to confirm or otherwise the efficacy of eliminating the key phrase "`is not null`" in joins in which it is redundant in the context of the *where* clause “`where COL is not null`”. This is described in detail in Section 4.4.3.4 where we gave a typical example (Example 4.4.9, page 120) which suggested that removal of this redundancy could halve the query cost.

6.13.1 Hypotheses

- Removing the redundant key phrase “`is not null`” from the where clause “`where COL is not null`” will increase query efficiency.
- The effect will be negligible below a certain threshold proportional to the number of rows returned by the query.

6.13.2 Method

In this experiment we submit batches of joins to the database, identical except that one batch has all *where* clauses containing the redundant “`is not null`” phrase while the other does not. As for the previous experiments, we measure the *ratio* of the optimized batch cost (redundant “`is not null`” eliminated) to the unoptimized batch cost (includes redundant “`is not null`” keyword). We reason the advantage of eliminating the phrase will be negligible when the time required to check the rows returned by the join is very small, relative to the total join processing time. For this reason:

- The independent variable in this case is the average number of rows returned per join: $Rows/Join$.
- The dependent variable is the cost ratio R_{com} as defined in Table 5.1 (page 139).

All results are for tables with number of rows $Rows = 500,000$. All joins consist of a single restriction defined by a single interval. All columns cited in query restrictions are indexed with a “normal” B-tree index.

Example 6.13.1. *The following is a typical join drawn from the batch used in these experiments:*

```
select distinct t1.ID id1, t2.ID id2, t1.COL8 c1
from   TAB6 t1, TAB1 t2
where  t2.COL2 = t1.COL2
and    t2.COL5 > 51547332
and    t2.COL5 < 52588692
and    t1.ID is not null
and    t2.ID is not null;
```

6.13.3 Results and Analysis

Figure 6.12(b) plots R_{com} versus $Rows/Query$. We first considered this optimization in Example 4.4.9 (page 120) where the SQL optimizer predicted a saving of around 50%. However, our experiments with batches of joins produced a consistent saving of just 5% across a wide range of $Rows/Query$.

6.13.4 Conclusion

Elimination of the redundant phrase "`is not null`" produces a measurable but very small increase in query efficiency of approximately 5%. The effect we measured was almost uniform across a wide range of $Rows/Join$. Although we have confirmed an efficiency gain, the magnitude of the gain is considerably less than the 50% gain suggested by the SQL optimizer itself which we report in Section 4.4.3.4 (page 120). We conjecture the reason for this is that the relatively small computational cost of checking all values are non-null is swamped by the caching of tabular data as the query batch proceeds. The hypotheses are nevertheless confirmed.

6.14 Query Restriction Introduction and Removal

The objective of this series of experiments is to investigate the efficacy of introducing additional predicates into queries, or eliminating them. This was described in detail in Sections 2.5.3 (page 34) and 2.5.2 (page 31) respectively. The extra predicates are generated as a consequence of a rule that has been discovered that correlates the values of an unindexed column of low selectivity and an indexed column of high selectivity. In the following experiments, the rule is of the form:

```
if COL20 = 'x' then COL1 between a and b;
```

In the above rule, `COL20` is the unindexed column of low selectivity while `COL1` is the indexed column with high selectivity. We expect queries restricted only on `COL20` to provoke a full table scan, since no index exists on this column. Conversely, an additional predicate restricted on `COL1` would be expected to provoke the SQL optimizer into consulting the index on `COL1`, thus avoiding a full table scan. This is *restriction introduction*.

It is equally possible to view the correlation captured by the rule above the other way around. That is, we discover a rule of the form:

```
if COL1 between p and q then COL20 = 'y';
```

Again, `COL20` is the unindexed column of low selectivity while `COL1` is the indexed column with high selectivity. When a query appears with restrictions on both `COL1` and COL20 and the restriction on COL20 can be inferred from the restriction on COL1 via the above rule, then we reason the restriction on COL20 can be ignored, simplifying the query. This is *restriction removal*.

For the purposes of comparison, we report on both *restriction introduction* and *restriction removal* simultaneously. In addition, we introduce another variable by considering the following two scenarios.

1. The column of low selectivity (COL20) is unindexed.

2. The column of low selectivity (COL20) is indexed using a *bitmap index* (Cyran, Lane & Polk 2005*c*).

The first scenario above is typically what researchers in SQO have in mind when restriction introduction and removal is described (Chomicki 2002, Lowden & Robinson 2002, Cheng et al. 1999).

The second scenario above recognises the fact that, in practice, it is most likely that the column cited in the rule pre-condition would be indexed with a bitmap index, which is particularly suited to columns of low selectivity (Chan 2005*f*). It turns out this has a positive effect on the query efficiency, as our results below in Section 6.14.3 show.

6.14.1 Hypotheses

- *Restriction introduction*: Adding a restriction on an indexed column of high selectivity will increase query efficiency.

- *Restriction removal*: Removing a restriction on an unindexed column of low selectivity will increase query efficiency.

- *Effect of bitmap*: The presence of a bitmap index on the unselective column will reduce the relative effectiveness of restriction introduction.

6.14.2 Method

1. We first establish a baseline with a batch of normal queries restricted only on COL20, the unindexed column with low selectivity. The following is a typical baseline query drawn from the batch used in these experiments:

```
select t.COL1, t.COL20
from   TAB1 t
where  t.COL20 = 'B';
```

2. The restriction introduction rule is then applied and an additional predicate generated for each normal query that restricts column `COL1`, which is indexed with a normal B-tree index. The modified queries are then re-submitted. The baseline query is modified in the following way:

```
select t.COL1, t.COL20
from   TAB1 t
where  t.COL20 = 'B'
and    t.COL1 between 46700008 and 46879992;
```

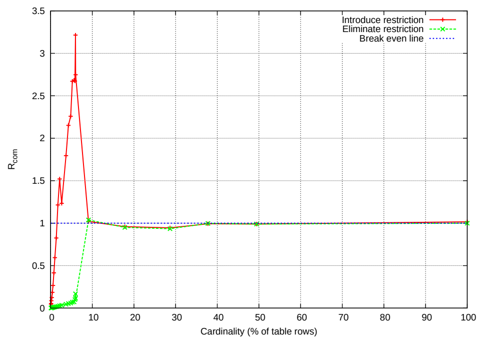
3. The restriction elimination rule is then applied and the restriction on `COL20` is removed, leaving only the restriction on `COL1`. The modified queries are then re-submitted. The query is modified in the following way:

```
select t.COL1, t.COL20
from   TAB1 t
where  t.COL1 between 46700008 and 46879992;
```

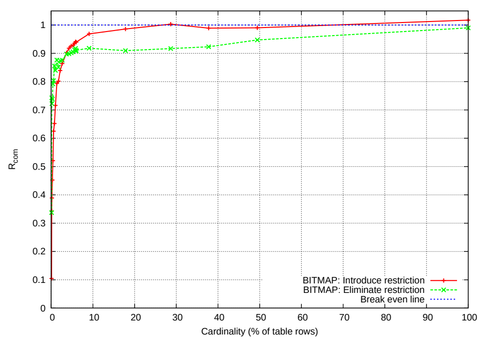
The independent variable is *Cardinality*, expressed as a percentage of the total number of table rows. The dependent variable is R_{com}, the combined metric cost ratio. Number of table rows is $750,000$.

6.14.3 Results and Analysis

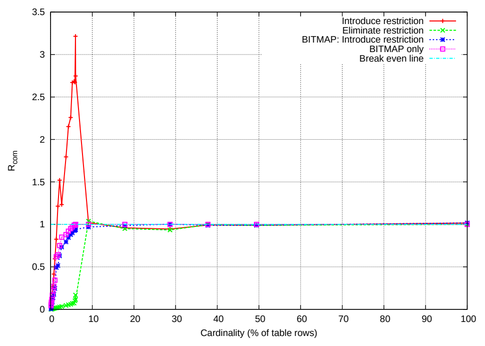
Consider Figure 6.13(a) which pictures the traditional scenario where a restriction on unindexed `COL20` is supplemented with an additional restriction on indexed `COL1` (*red* line). Our results repeatedly confirmed that once the cardinality of the result set exceeds a certain threshold (approximately 10% for our target tables), this provokes the SQL optimizer to opt for a full table scan. This is because the SQL optimizer judges the resources required to first consult the index on `COL1` would exceed the resources required to perform a full table scan. Therefore, no advantage is conferred by the additional predicate in this situation, which covers approximately 90% of the range of query selectivities from $10 - 100\%$. In fact, there is only a small range of query selectivities from approximately $0 - 2\%$ where the additional predicate confers an advantage. The results for query selectivities in the range $2 - 10\%$ are initially counter-intuitive. The additional predicate, far from reducing query cost, provokes the optimizer into choosing a path which is up to 3 times more costly than a full table scan. We studied the execution plans produced by the SQL optimizer in these circumstances and repeatedly confirmed the optimizer judges it worthwhile to consult the index on `COL1` for cardinalities under approximately 10%. In fact, the actual execution time turns out in these circumstances to be significantly longer. We



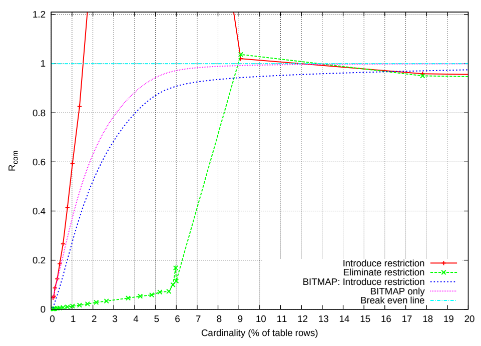
(a) No index on COL20



(b) Bitmap index on COL20



(c) Comparing efficiency gain



(d) Comparing efficiency gain: cardinality 0 – 20%

Figure 6.13: **Query restriction introduction and elimination**: The traditional scenario is pictured in Figure 6.13(a) where a restriction on unindexed COL20 is supplemented with an additional restriction on indexed COL1 (*red* line). Conversely, the restriction on unindexed COL20 can be removed leaving only the restriction on COL1 (*green* line). A more realistic scenario is pictured in Figure 6.13(b). Here the low selectivity column COL20 is indexed with a bitmap. In Figures 6.13(c) and 6.13(d) we compare the efficiency gain produced with and without a bitmap index on COL20 by combining the results from Figures 6.13(a) and 6.13(b). Also included is the effect of simply adding the bitmap index to COL20.

emphasize that our optimizer settings are the standard, default setting for the Oracle RDBMS and optimizer statistics were always "fresh".

Consider the *green* line of Figure 6.13(a) which pictures the traditional scenario where a restriction on unindexed COL20 has been eliminated. Again, once the cardinality of the result set exceeds a certain threshold (approximately 10% for our target tables), this provokes the SQL optimizer to opt for a full table scan. However, we obtain a useful increase in query efficiency across the selectivity range 0 – 10%, a far better result than for restriction introduction.

Now consider Figure 6.13(b) where we display results for exactly the same query batches, but this time we have placed a bitmap index on COL20 (*red* line). The primary beneficial effect of the bitmap is that the SQL optimizer now accurately predicts the cardinality of the result set. We studied the execution plans produced by the SQL optimizer in these circumstances and repeatedly confirmed the optimizer firstly "pre-selects" using the bitmap index, then scans using the B-tree index[3]. The

[3]The details of the execution plan are beyond the scope of this analysis. In fact, the Oracle

cost of scanning the two indexes is not the same; scanning the bitmap index is typically judged by the SQL optimizer to be a factor of 10 less costly than scanning the B-tree index. The overall effect of using both indexes is that query efficiency is increased up to a selectivity of approximately 10% when the SQL optimizer orders a full table scan regardless of the indexes.

Consider the *green* line of Figure 6.13(b) which pictures the scenario where a restriction on unindexed `COL20` has been eliminated. Comparing the efficiency gain with the restriction introduction case, we again see restriction elimination yields a better result, although for query selectivity from 10 – 50% the gain is less than 10%.

Consider Figures 6.13(c) and 6.13(d). These display exactly the same results as Figures 6.13(a) and 6.13(b) but plotted on the same graph for ease of comparison. Also included is the efficiency gain produced by simply placing the bitmap on `COL20` without the addition or removal of restrictions. The baseline in each case is the cost for queries restricted only on `COL20` without a bitmap. Considering the magnified results displayed in Figure 6.13(d), we see that in the case of the bitmapped `COL20`, the major efficiency gain is produced simply by the presence of this index (*pink* line). A small additional gain is produced by introducing a restriction (*blue* line) and again the biggest advantage is realised by eliminating a restriction (*green* line).

6.14.4 Conclusion

We now consider the experimental results with respect to the hypotheses above in Section 6.14.1.

- *Restriction introduction*: The traditional scenario of restriction introduction is highly restricted in its usefulness. In order to provoke the SQL optimizer into efficiently utilising the index associated with the additional predicate, the optimizer must be able to *accurately* predict the cardinality of the result set. Our results repeatedly showed it was quite easy to “fool” the optimizer into making the wrong decision. This is most dramatically shown in Figure 6.13(a) for query selectivities between 2 – 10% where the optimizer has significantly underestimated the cost of index scanning. Restriction introduction increased query efficiency only for query selectivities in the narrow range of approximately 0 – 2%. For query selectivities greater than approximately 10%, the SQL optimizer orders a full table scan and the extra predicate has no effect. The hypothesis is therefore confirmed only for the lowest query selectivities in the narrow range of approximately 0 – 2%.

optimizer uses a *hash join* to efficiently correlate the information contained in these two indexes and it is primarily this that leverages the advantage of using both indexes.

- *Restriction removal*: The removal of the redundant restriction on the unselective column was a more effective strategy than adding a restriction on the selective column, regardless of the presence of a bitmap index on the unselective column. Restriction removal increased query efficiency across approximately the range for which the SQL optimizer does not order a full table scan ($0-10\%$ in our experiments). The hypothesis is therefore confirmed for query selectivities in the range of approximately $0-10\%$.

- *Effect of bitmap*: The primary beneficial effect of adding a bitmap index to unselective column `COL20` is that it allows the SQL optimizer to accurately predict the cardinality of the result set. This is clearly shown in Figure 6.13(d). In fact the benefit of the bitmap index alone on the unselective column is almost as great as when this is combined with the additional restriction on the selective column with the B-tree index. The hypothesis is therefore *not* confirmed because in fact the bitmap index enhances the restriction introduction strategy.

6.15 Joins Restriction Introduction and Removal

The objective of this series of experiments is to investigate the efficacy of introducing additional predicates into joins, or eliminating them. These experiments repeat the scenario described in detail above for queries in Section 6.14 but with restrictions applied to one half of the equi-join.

6.15.1 Hypotheses

- *Restriction introduction*: Adding a restriction on an indexed column of high selectivity will increase join efficiency.

- *Restriction removal*: Removing a restriction on an unindexed column of low selectivity will increase join efficiency.

- *Effect of bitmap*: The presence of a bitmap index on the unselective column will reduce the relative effectiveness of restriction introduction.

6.15.2 Method

1. We first establish a baseline with a batch of normal joins where one table is restricted only on `COL20`, the unindexed column with low selectivity. The following is a typical baseline join drawn from the batch used in these experiments:

```
select t1.COL1, t1.COL20, t2.COL1, t2.COL20
from   TAB1 t1, TAB2 t2
where  t1.COL5 = t2.COL5
and    t1.COL20 = 'B';
```

2. The restriction introduction rule is then applied and an additional predicate generated for each normal join that restricts column `COL1`, which is indexed with a normal B-tree index. The modified joins are then re-submitted. The baseline join is modified in the following way:

   ```
   select t1.COL1, t1.COL20, t2.COL1, t2.COL20
   from   TAB1 t1, TAB2 t2
   where  t1.COL5 = t2.COL5
   and    t1.COL20 = 'B'
   and    t1.COL1 between 46700008 and 46879992;
   ```

3. The restriction elimination rule is then applied and the restriction on `COL20` is removed, leaving only the restriction on `COL1`. The modified joins are then re-submitted. The join is modified in the following way:

   ```
   select t1.COL1, t1.COL20, t2.COL1, t2.COL20
   from   TAB1 t1, TAB2 t2
   where  t1.COL5 = t2.COL5
   and    t1.COL1 between 46700008 and 46879992;
   ```

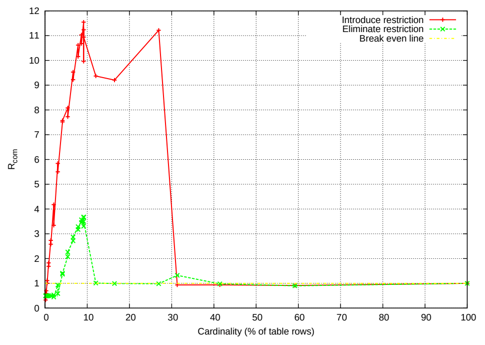
The independent variable is *Cardinality*, expressed as a percentage of the total number of table rows. The dependent variable is R_{com}, the combined metric cost ratio. Number of table rows is $750,000$.

6.15.3 Results and Analysis

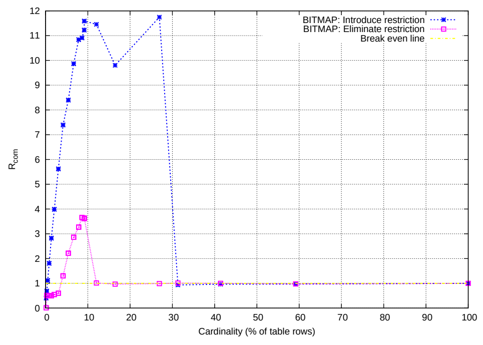
We first note that results for joins with and without a bitmap index on the unselective `COL20` are almost identical, so no significant advantage (or disadvantage) results from this index. For join selectivities between $0-30\%$ the SQL optimizer has significantly underestimated the cost of consulting the B-tree index on `COL1`. The restriction removal strategy produces a better result than restriction introduction, which is analogous to the result reported for queries above in Section 6.14.

6.15.4 Conclusion

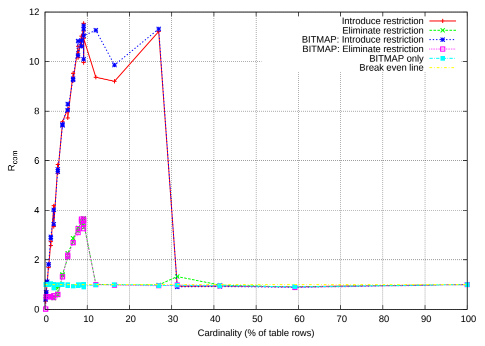
We now consider the experimental results with respect to the hypotheses above in Section 6.15.1.



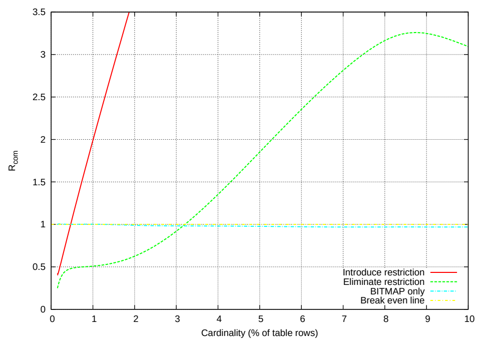
(a) No index on `COL20`



(b) Bitmap index on `COL20`



(c) Comparing efficiency gain



(d) Comparing efficiency gain: cardinality 0 – 10%

Figure 6.14: **Join restriction introduction and elimination**: Restriction introduction does not benefit the equi-joins tested in our experimental batches, apart from a narrow range of the most selective joins. Restriction removal produces a better result, enhancing join efficiency for selectivities in the range 0 – 3%. The presence of a bitmap index on the unselective column has no impact on overall join efficiency.

- *Restriction introduction*: Adding additional restrictions to the equi-joins that we tested is not a promising strategy to increase query efficiency. Although we measured an efficiency gain for joins with very low selectivity, we found the SQL optimizer significantly underestimated the cost of consulting the B-tree index on the column cited in the introduced restriction. This is the same result we observed as for simple queries, only the effect is more pronounced. The hypothesis is not confirmed except for a very narrow range of join selectivities where the SQL optimizer is able to accurately predict the cardinality of the result set.

- *Restriction removal*: Removing the redundant restriction on the unselective column yielded a modest gain in join efficiency for joins with selectivities in the range 0 – 3%. Restriction removal is a more promising strategy than restriction introduction, a result that we also observed for simple queries. The hypothesis is confirmed for a narrow range of joins with very low selectivity.

- *Effect of bitmap*: The presence or absence of a bitmap index on the unselective column has no effect on overall join efficiency. The hypothesis is not confirmed.

6.16 Summary

The main objective of this Chapter is to present the results of a series of experiments whose overall aim is to demonstrate the effectiveness of the different types of SQO we described first in Chapter 2 and expanded upon in Chapter 4. A critical feature of our experiments is that the database schema we employ closely simulates typical schemas found in real RDBMS environments. In particular, we give results for queries against large tables with columns that are *indexed*. To our knowledge, these are the first comprehensive empirical results which demonstrate SQO for queries and equi-joins, in the presence of standard indexes.

The main contributions of this Chapter include the following.

- In the case of *unindexed tables*, we confirm that the gain in query efficiency increases linearly with increasing probability of an unsatisfiable query. The gain in query efficiency is accurately predicted by the idealised cost model of Section 5.4.2.3 and is almost independent of table size. Crucially, our cost model sets an upper bound for the amount of optimization we can expect from detecting unsatisfiable queries. That is, even if the cost of detecting unsatisfiable queries is negligible (and it is not), the maximum amount of optimization is irrevocably determined by the prevalence of unsatisfiable queries (Section 6.3).

- In the case of *indexed tables*, we confirm that the gain in query efficiency increases linearly with increasing probability of an unsatisfiable query. The gain in query efficiency is accurately predicted by the idealised cost model of Section 5.4.2.3 and is almost independent of table size (Section 6.4).

- In the case of *indexed tables*, we confirm that the gain in join efficiency increases linearly with increasing probability of an unsatisfiable join. The gain in join efficiency is accurately predicted by the idealised cost model of Section 5.4.2.3 and is almost independent of table size (Section 6.7).

- Although the extra computational costs incurred by the semantic query preprocessor are relatively small (compared to the computational cost of processing the query with the normal SQL optimizer) for unindexed tables, these costs cannot be discounted when tables are sensibly indexed. For our prototype semantic optimizer, the probability of an unsatisfiable query needs to be approximately 10% (for queries) or 20% (for equi-joins) to break even (Sections 6.3, 6.4 and 6.7).

- In the case of *indexed tables,* we confirm for both queries and joins that as *query difficulty* (as defined in Section 5.3.2) increases, so does the cost of semantically preprocessing the queries. An increasing proportion of unsatisfiable queries is required to break even. However, for modest numbers of restrictions per query ($R/Q \approx 5$) and intervals per restriction ($I/R \approx 5$) our prototype semantic optimizer breaks even when the probability of an unsatisfiable query (P) reaches ≈ 0.2 (Sections 6.5, 6.6, 6.8 and 6.9).

- We confirm that, for both queries and joins, it is worthwhile eliminating the redundant keyword "`distinct`" from the select clause "`select distinct`". However, averaged out over a large number of queries(joins), the net saving made was significantly less than predicted by the SQL optimizer (Sections 6.10 and 6.11).

- With regard to eliminating the redundant key phrase "`is not null`" from the where clause "`where COL is not null`", for queries we were unable to show a consistent average efficiency gain. For joins, we measured a consistent average gain of just 5%. Both of these results are substantially less than predicted by the SQL optimizer (Sections 6.12 and 6.13). Comparing this result with the result for eliminating the redundant keyword "`distinct`" above, we may conclude that the cost of performing the redundant sort is more significant than the cost of checking values are non-null.

- We consistently observed that for both queries and joins, the *restriction removal* strategy was significantly more effective than *restriction introduction.* With standard SQL optimizer settings, the Oracle RDBMS tended to underestimate the cost of B-tree index scanning on the introduced predicate, resulting in sub-optimal execution paths. Restriction removal was seen to be effective for queries and joins with low selectivity; i.e., where the cardinality of the result set was low compared to total table size. These results strongly suggest an effective SQO strategy for tables of the type we have studied would be to look for rules that allow restrictions on columns of low selectivity to be eliminated. (Sections 6.14 and 6.15).

Chapter 7

Related Work and Extensions

7.1 Introduction

The interval algebra we develop in Chapter 3 is central to our research. It is the basis of our reasoning engine which we utilise to implement SQO. So far in this thesis, we have primarily focussed on the operations conjunction, disjunction and negation with intervals and interval lists. However, in this chapter we focus on the use of intervals in other areas of research. We highlight the versatility of the interval algebra we have implemented and how it can be utilised in a variety of areas with very little extension or modification. We begin by examining *arithmetic with intervals* over the Real numbers. We then examine *temporal intervals* where the intervals are conceived to specifically represent periods of time.

The remainder of this chapter is set out as follows.

- We firstly focus on interval arithmetic, showing how the four basic arithmetic operations of orthodox interval arithmetic can be meaningfully extended by subtly altering the treatment of the two infinities and zero, while extending the range of intervals to include both inclusive and exclusive limits (Section 7.2).

- We then show how arithmetic with intervals can be easily extended to arithmetic with interval lists and how this can be used to extend interval division (Section 7.3).

- We study the special case of temporal intervals, describing how the Allen interval algebra (Allen 1983) is fully expressible within our own interval algebra and how it can be meaningfully extended to include both inclusive and exclusive limits (Section 7.4).

7.2 Interval Arithmetic

The fundamental idea of interval arithmetic is that calculations are performed on pairs of *intervals*, rather than pairs of numbers (Clemmesen 1984). Applying an arithmetic operation, such as addition, to a pair of intervals yields an interval containing all numbers resulting from applying the same operation to all pairs of numbers from the two intervals (Hickey, Ju & Emden 2001, Clemmesen 1984). The intervals referred to in interval arithmetic are conventionally defined on the Real numbers by their endpoints in the following way (Muñoz & Lester 2005, Walster 2000).

$$[a, b] = \{x \in R : a \leq x \leq b\}$$

The numbers $a, b \in R$ form the left and right bounds of the interval [1]. If $a > b$ the interval is empty [2] and the interval $[a, a]$ denotes the point a (Muñoz & Lester 2005, Hickey et al. 2001).

The intervals that result from the application of the four arithmetic operations addition (+), subtraction (−), multiplication (×) and division (÷) are given by the following formulae (Muñoz & Lester 2005, Hickey et al. 2001, Walster 2000). A is the interval $[a, b]$ and B is the interval $[c, d]$ where $a, b, c, d \in R$.

$$\begin{aligned}
A + B &= [a + c, b + d] \\
A - B &= [a - d, b - c] \\
A \times B &= [min\,(S_1)\,, max\,(S_1)] && \text{where } S_1 \in \{ac, ad, bc, bd\} \\
A \div B &= [min\,(S_2)\,, max\,(S_2)] && \text{where } S_2 \in \{a/c, a/d, b/c, b/d\}
\end{aligned}$$

For interval multiplication, the left and right bounds are chosen from S_1, the set of products $\{ac, ad, bc, bd\}$. For interval division, the left and right bounds are chosen from S_2, the set of quotients $\{a/c, a/d, b/c, b/d\}$. Interval division is not defined when the interval divisor (B in the formula above) includes zero.

The basic interval arithmetic operations on the Real numbers defined above are *closed*, provided interval divisors containing zero are disallowed for division (Walster 2000). They can also be extended to handle *minus infinity* and *plus infinity*. In this case the set of numbers is called the *Extended Reals* which is defined to be the set $R \cup \{^-\infty, ^+\infty\}$. The following extra definitions are applied to intervals on the Extended Reals (Hickey et al. 2001).

$$\begin{aligned}
[^-\infty, b] &= \{x \in R : x \leq b\} \\
[a, ^+\infty] &= \{x \in R : a \leq x\} \\
[^-\infty, ^+\infty] &= R
\end{aligned}$$

7.2.1 Differences in our interval implementation

We now describe several important differences between our implementation of intervals and the orthodox implementation described above.

We first introduced the idea of incorporating *minus infinity* and *plus infinity* into our generic data type T in Section 3.2.5 (page 47) and the Extended Reals was the model we had in mind. We have implemented a version of the four interval arithmetic operations described above for the numeric data type and these incorporate the handling of *minus infinity* and *plus infinity* in an analogous manner to that

[1]In this thesis we use the terms "left" and "right" respectively throughout, rather than the more orthodox "upper" and "lower" to avoid confusion with our definitions for conjunction and disjunction in Chapter 3.

[2]We use this as our definition of the null interval. See Section 3.5.1.2, page 61.

described by, for example, (Hickey et al. 2001, Walster 2000).

- However, our implementation is more general in that we use *minus infinity* and *plus infinity* with data types other than numeric. When we use these terms with non-numeric data types, we intend only to convey the idea of a value which would always be the first(last) value were it added to any ordered list of values. While our implementation does not in general allow arithmetic on non-numeric data types[3], it nevertheless recognises *minus infinity* and *plus infinity* as special constants and these are dealt with in a meaningful way such that the conjunction, disjunction and negation operations on interval data are implemented correctly.

- Our interval implementation differs in its treatment of *limits*. Limits were first described in Section 3.3 (page 48) and we deliberately treated the four limits "(,) , [,]" as *operators* that may only operate on values to produce *bounds* (Section 3.4, page 51). The outcome for intervals is that we allow both inclusive and exclusive bounds. So rather than just the interval $[a, b]$ we allow the four intervals: (a, b), $[a, b)$, $(a, b]$, $[a, b]$. This allows us to meaningfully apply the arithmetic operators to a wider set of intervals.

We now describe how we extend interval arithmetic to include intervals with both inclusive and exclusive bounds. For clarity we initially consider only the Real numbers and neglect the special cases involving *minus infinity* and *plus infinity*. We then give our own algorithm for *extended interval arithmetic* and show how *minus infinity* and *plus infinity* can be effortlessly incorporated.

7.2.2 Extending Interval Addition

We gave the formula for interval addition in Section 7.2 (page 194) above as:

$$A + B = [a + c, b + d]$$

where A is the interval $[a, b]$ and B is the interval $[c, d]$ and $a, b, c, d \in R$. However, if we relax the requirement that the left and right bounds must be inclusive, we still obtain meaningful results with clear semantics. Figure 7.1 (page 197) sets out all the possibilities that may arise in this extended interval addition. We now provide an informal proof for the results shown in Figure 7.1.

Theorem 7.2.1. ***Extended Interval Addition***: *The result of applying the addition operator to intervals comprising both inclusive and exclusive bounds is given by the table of Figure 7.1.*

[3]See Section 7.4 below (page 215) for examples of interval arithmetic with the *date* data type.

A_a	B_c	$A_a + B_c$		A_b	B_d	$A_b + B_d$
[a	[c	[a+c		b]	d]	b+d]
[a	(c	(a+c		b]	d)	b+d)
(a	[c	(a+c		b)	d]	b+d)
(a	(c	(a+c		b)	d)	b+d)

Figure 7.1: **Extended interval addition A + B**: When the requirement is relaxed that the left and right bounds must be inclusive, interval addition still yields meaningful results with clear semantics. This table sets out the four possibilities for each of the left and right bounds. The result reduces to the orthodox interval addition formula when both intervals are composed of inclusive bounds.

***Proof**: We use the right arrow symbol "→" in the proof below to mean "approaches" in the sense of the Calculus of Limits (Anton 1984). Consider the result for orthodox interval addition:*

$$\begin{aligned} A &= [a, b] \\ B &= [c, d] \\ A + B &= [a + c, b + d] \\ &= [x, y] \qquad \textit{where } x = a + c,\ y = b + d \end{aligned}$$

Consider how this result must change if the first interval A is $(a, b]$. We replace the exclusive left bound with an inclusive left bound in the following way:

$$\begin{aligned} A &= (a, b] \\ &= [a', b] \qquad \textit{where } a' > a \end{aligned}$$

Adding the two intervals yields the following:

$$A + B = [a' + c, b + d]$$

Now consider what happens as a' approaches a from the right:

$$\lim_{a' \to a^+} (a' + c) = x$$

Therefore we may write:

$$\begin{aligned} A + B &= (x, b + d] \\ &= (a + c, b + d] \end{aligned}$$

An analogous argument can be made for the other cases depicted in Figure 7.1. Inspection of Figure 7.1 yields the simple rule that if one or more of the bound limits is exclusive, "(" or ")", then the result of the addition is also exclusive.

7.2.3 Extending Interval Subtraction

We gave the formula for interval subtraction in Section 7.2 (page 194) above as:

$$A - B = [a - d, b - c]$$

where A is the interval $[a, b]$ and B is the interval $[c, d]$ and $a, b, c, d \in R$. However, if we relax the requirement that the left and right bounds must be inclusive, we still obtain meaningful results with clear semantics. Figure 7.2 sets out all the possibilities that may arise in this extended interval subtraction. The proof of the results

A_a	B_d	A_a - B_d		A_b	B_c	A_b - B_c
[a	d]	[a-d		b]	[c	b-c]
[a	d)	(a-d		b]	(c	b-c)
(a	d]	(a-d		b)	(c	b-c)
(a	d)	(a-d		b)	(c	b-c)

Figure 7.2: **Extended interval subtraction A – B**: When the requirement is relaxed that the left and right bounds must be inclusive, interval subtraction still yields meaningful results with clear semantics. This table sets out the four possibilities for each of the left and right bounds. The result reduces to the orthodox interval subtraction formula when both intervals are composed of inclusive bounds.

depicted in Figure 7.2 proceeds in an analogous fashion to the proof sketched above in Section 7.2.2 for Extended Interval Addition. Again, we arrive at the simple rule that if one or more of the bound limits is exclusive, "(" or ")", then the result of the subtraction is also exclusive.

We defer descriptions of how we extend interval multiplication and division until Sections 7.2.5 (page 203) and 7.3.2 (page 210) respectively.

7.2.4 Algorithm for Extended Interval Arithmetic

We now describe a general algorithm for *extended interval arithmetic* (EIA) on the Real numbers. We first describe the major innovations of the algorithm. We then state the algorithm itself in Section 7.2.4.1 (page 199). This is followed in Section 7.2.4.2 (page 201) by a description of how the algorithm must be restricted to maintain its validity when, for example, we allow *minus infinity* and *plus infinity* to be one or both of the operands.

The major innovations of this algorithm are:

- We allow arithmetic with intervals that include both inclusive and exclusive bounds. For example, the following calculation is allowed:

$$[3, 5) + (100, 105] = (103, 110)$$

- We apply the arithmetic operation *at the bound level*. That is, we never simply apply the arithmetic operator to the two values. Instead, the *limit* which is associated with each *value* is always included in the calculation.
- We allow a restricted occurrence of *minus infinity* and *plus infinity*, which we treat as special constants and which, we argue, enhance the expressiveness of interval arithmetic.
- We allow division by values that *approach* zero. This facilitates calculation with a wide range of cases that would be disallowed by orthodox interval division.

For clarity, we first reiterate some helpful definitions from Chapter 3. Table 7.1 lists the various types of numeric bounds which we refer to and their equivalent definitions in set notation.

Name	Bound	Set
Inclusive left bound	$[a$	$\{x \in R : a \leq x\}$
Inclusive right bound	$b]$	$\{x \in R : x \leq b\}$
Exclusive left bound	$(a$	$\{x \in R : a < x\}$
Exclusive right bound	$b)$	$\{x \in R : x < b\}$
Inclusive zero bound	$[0$	$\{x \in R : 0 \leq x\}$
Inclusive zero bound	$0]$	$\{x \in R : x \leq 0\}$
Exclusive zero bound	$(0$	$\{x \in R : 0 < x\}$
Exclusive zero bound	$0)$	$\{x \in R : x < 0\}$
Inclusive infinite bound	$[{}^{-}\infty$	$\{x \in R : {}^{-}\infty \leq x\}$
Inclusive infinite bound	${}^{+}\infty]$	$\{x \in R : x \leq {}^{+}\infty\}$
Exclusive infinite bound	$({}^{-}\infty$	$\{x \in R : {}^{-}\infty < x\}$
Exclusive infinite bound	${}^{+}\infty)$	$\{x \in R : x < {}^{+}\infty\}$

Table 7.1: **Some useful numeric bounds** defined over the Real numbers and their equivalent definitions in set notation.

7.2.4.1 Statement of EIA algorithm

We first give an algorithm to determine if the bound which results from a given calculation is inclusive or exclusive. This algorithm determines only this information; it does not determine whether the bound is a left bound or a right bound.

Algorithm 7.2.1. ***Inclusive or Exclusive Bound****: Let B_1 and B_2 be any two bounds, as we have defined then in Definition 3.4.1 (page 51). Bound B_1 comprises a limit l_1 and a value v_1 such that $B_1 = l_1 v_1$. Similarly, bound B_2 comprises a limit l_2 and a value v_2 such that $B_2 = l_2 v_2$ Let "$\otimes$" denote any of the four interval arithmetic operations: $\otimes \in \{+, -, \times, \div\}$. Then:*

If B_1 is an inclusive zero bound then

```
        $B_1 \times B_2$ is inclusive;
        $B_1 \div B_2$ is inclusive;
    else if $B_2$ is an inclusive zero bound then
        $B_1 \times B_2$ is inclusive;
        $B_1 \div B_2$ is undefined;
    else if $B_1$ is inclusive and $B_2$ is inclusive then
        $B_1 \otimes B_2$ is inclusive;
    else $B_1 \otimes B_2$ is exclusive;
```

In fact it is a little easier in this case to state the algorithm informally: If both bounds are inclusive then the result is also inclusive, otherwise the result is exclusive. The exceptions are multiplication with an inclusive zero bound, which always results in an inclusive zero bound and division where the dividend is an inclusive zero bound, which always results in an inclusive zero bound.

We now state the EIA algorithm itself. When we refer to the "orthodox formulae" for interval arithmetic we mean precisely the formulae we state in Section 7.2 (page 194) for the four basic arithmetic operations.

Algorithm 7.2.2. ***Extended Interval Arithmetic****: Consider an arbitrary interval $I = B_a, B_b$ comprising left bound B_a and right bound B_b. Similarly, interval $J = B_c, B_d$ comprises left bound B_c and right bound B_d. Each bound in turn comprises a limit and a value: $B_i = l_i v_i$ where $i \in \{a, b, c, d\}$. Let "$\otimes$" denote any of the four interval arithmetic operations: $\otimes \in \{+, -, \times, \div\}$. When one of these operators is applied to intervals I and J, it is always the case that one operand is supplied by I, which we will indicate with the subscript "i", while the other is supplied by J, which we will indicate with the subscript "j". Then $I \otimes J$ is calculated as follows:*

1. *Apply the orthodox formula for interval arithmetic to the appropriate value pairs: $v_i \otimes v_j$*

2. *For each pair of operands in 1, apply Algorithm 7.2.1 using limits l_i and l_j to decide if the answer is inclusive or exclusive.*

3. *In the case of multiplication and division, discard products(quotients) that are classified as* undefined*.*

4. *In the case of multiplication and division, choose the minimum value v_{min} and apply the inclusive or exclusive left limit, as dictated by 2. Similarly, choose the maximum value v_{max} and apply the inclusive or exclusive right limit, as dictated by 2.*

Point 3 in Algorithm 7.2.2 above foreshadows Figure 7.3 (page 201) and is explained in Section 7.2.4.2 below. Examples of discarding undefined products and quotients are given in Sections 7.2.5 and 7.3.2.

7.2.4.2 Restrictions applied to EIA algorithm

Operator	Undefined Operation	Result
+, -, x, ÷	*either operand* $[^{\pm}\infty]$	-
÷	B ÷ [0]	-
x	(0 x $(^{-}\infty$	un^{-}
x	(0 x $^{+}\infty)$	un^{+}
x	0) x $(^{-}\infty$	un^{+}
x	0) x $^{+}\infty)$	un^{-}
÷	$(^{-}\infty$ ÷ $(^{-}\infty$	un^{+}
÷	$(^{-}\infty$ ÷ $^{+}\infty)$	un^{-}
÷	$^{+}\infty)$ ÷ $(^{-}\infty$	un^{-}
÷	$^{+}\infty)$ ÷ $^{+}\infty)$	un^{+}
÷	(0 ÷ (0	un^{+}
÷	(0 ÷ 0)	un^{-}
÷	0) ÷ (0	un^{-}
÷	0) ÷ 0)	un^{+}

Figure 7.3: **Undefined operations for Extended Interval Arithmetic**: The table lists the arithmetic operations for which are undefined when we carry out extended interval arithmetic (EIA). The first row prohibits any operations with the inclusive infinities. The second row prohibits any division by an inclusive zero ("**B**" denotes any bound). The remaining rows list the exclusive cases involving zero and infinity which are undefined. For multiplication, the operation with operands reversed is omitted, since multiplication is commutative. In the *Result* column, "un^{+}" denotes "undefined positive"; "un^{-}" denotes "undefined negative".

We apply the following restrictions to our EIA algorithm.

1. *Restrictions across all four arithmetic operations*:

   - Neither the left nor the right bound of either interval may contain an inclusive infinity. Therefore the bounds $[^{-}\infty$, $[^{+}\infty$, $^{+}\infty]$, $^{-}\infty]$ are all disallowed in arithmetic calculations. This ensures that we do not attempt to calculate with either *minus infinity* or *plus infinity* itself, which is undefined (Muñoz & Lester 2005, Hickey et al. 2001). However, we do facilitate calculations where the parameters *approach minus infinity* or *plus infinity*.
   - The exclusive bounds $^{-}\infty)$ and $(^{+}\infty$ are disallowed since they imply the existence of values to the left(right) of minus(plus) infinity, a situation which is logically excluded by the Definitions 3.2.1 and 3.2.2 (page 47).

2. *Additional restrictions on addition*:

   - There are no additional restrictions required for interval addition. This (perhaps counter-intuitive) conclusion arises because we do not have to consider addition between exclusive infinities of opposite sign, such as $+\infty) + (-\infty$. Although this sum is in fact undefined, it can never arise since the formula for interval addition sums only left bounds with left bounds and right bounds with right bounds.

3. *Additional restrictions on subtraction*:

   - There are no additional restrictions required for interval subtraction. This is because subtractions such as $(-\infty - (-\infty$ between exclusive infinities of the same sign never arise, by virtue of the formula for interval subtraction which only requires subtraction of right bounds from left bounds and vice versa.

4. *Additional restrictions on multiplication*:

   - Multiplication between exclusive zero bounds and exclusive infinite bounds is undefined. So $(0 \times +\infty)$ is undefined, as is $(-\infty \times 0)$. The most we can say about products with these particular bounds is to predict their sign. For example, $(0 \times +\infty)$ must yield a positive bound, since both operands are undeniably positive. We indicate this with the symbol un^{+} . Similarly, $0) \times +\infty)$ must yield a negative bound since $0)$ denotes a negative value while $+\infty)$ is positive. We indicate this with the symbol un^{-} . This is described in more detail below in Section 7.2.5 (page 203).

5. *Additional restrictions on division*:

   - We do not allow division by inclusive zero. However, we *do* allow division by *a bound that approaches zero*, that is, the exclusive zero bounds $0)$ and $(0$ [4]. This is described in more detail below in Section 7.3.2 (page 210).
   - We do not allow division when both dividend and divisor are exclusive zero bounds. Therefore $(0 \div (0$ and $0) \div (0$ are both undefined. Nevertheless, we can predict the sign of these quotients using exactly the same reasoning as for the undefined multiplications described above. For example, $0) \div 0)$ must be positive since both operands are negative. We use the symbol un^{-} to denote an undefined quotient which nevertheless must be negative and un^{+} to denote an undefined quotient which nevertheless must be positive.
   - We do not allow division between exclusive infinite bounds. For example, $+\infty) \div +\infty)$ is undefined, as is $(-\infty \div +\infty)$. Nevertheless, we

[4]provided the dividend is not itself an exclusive zero bound. See next bullet point.

can predict the sign of these quotients using exactly the same reasoning as for the undefined multiplications described above. For example, +∞) ÷ +∞) must be positive since both operands are positive.

A full list of undefined operations is listed in Figure 7.3 (page 201).

Since there are no extra restrictions needed for interval addition and subtraction, we can now provide a full definition for these two operations which includes both inclusive and exclusive bounds and the exclusive infinities. This is shown in Figure 7.4.

‹a	*‹c*	*‹a+‹c*		*b›*	*d›*	*b›+d›*
‹a	‹c	‹ a+c		b›	d›	b+d ›
‹a	(-∞	(-∞		b›	+∞)	+∞)
(-∞	‹c	(-∞		+∞)	d›	+∞)
(-∞	(-∞	(-∞		+∞)	+∞)	+∞)

(a) Interval addition with (-∞ and +∞).

‹a	*d›*	*‹a - d›*		*b›*	*‹c*	*b› - ‹c*
‹a	d›	‹ a-d		b›	‹c	b-c ›
‹a	+∞)	(-∞		b›	(-∞	+∞)
(-∞	d›	(-∞		+∞)	‹c	+∞)
(-∞	+∞)	(-∞		+∞)	(-∞	+∞)

(b) Interval subtraction with (-∞ and +∞).

Figure 7.4: **Extended interval addition and subtraction with** (-∞ **and** +∞): The tables show the outcome for interval addition and subtraction, including when one or both of the operands is an exclusive infinite bound. The left angle bracket "⟨" denotes either "(" or "[". Similarly, the right angle bracket "⟩" denotes either ")" or "]". Algorithm 7.2.1 is applied to determine if the bounds are inclusive or exclusive.

7.2.5 Extending Interval Multiplication

We now show how orthodox interval multiplication can be extended using Algorithm 7.2.2, having regard for the restrictions which we describe above in Section 7.2.4.2. We proceed by considering several examples which illustrate how we resolve difficult cases. We refer to Figure 7.5 (page 204) where we summarise how we calculate the answer for multiplication of intervals, including the "difficult" cases which involve the two infinities and zero.

We begin with the following Example 7.2.1 which illustrates the simple case where one bound is an exclusive infinity.

Example 7.2.1. *Consider Figure 7.6 which shows interval multiplication where the first right bound is* +∞) . *Therefore the products* **bc** *and* **bd** *have* +∞) *as one of their operands. We read the result from Figure 7.5(a) which in both cases is* +∞) . *Algorithm 7.2.1 is then applied to decide if the bound is inclusive* (***i***) *or exclusive* (***e***)*. We then identify the minimum which, by the interval multiplication formula, must be the left bound. Similarly, the maximum must be the right bound.*

In Example 7.2.2 we show what happens when one of the four products that are required for interval multiplication is undefined. We explain how the undefined case

		B_{La}					B_{Rb}				
	X	$(^{-}\infty$	$\langle^{-}a$	[0	(0	$\langle^{+}a$	$^{-}b\rangle$	0)	0]	$^{+}b\rangle$	$^{+}\infty)$
B_{Lc}	$(^{-}\infty$	$^{+}\infty)$	$^{+}\infty)$	[0]	un^{-}	$(^{-}\infty$	$^{+}\infty)$	un^{+}	[0]	$(^{-}\infty$	$(^{-}\infty$
	$\langle^{-}c$	$^{+}\infty)$	$^{+}\langle ac\rangle$	[0]	0)	$^{-}\langle ac\rangle$	$^{+}\langle bc\rangle$	(0	[0]	$^{-}\langle bc\rangle$	$(^{-}\infty$
	[0	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]
	(0	un^{-}	0)	[0]	(0	(0	0)	0)	[0]	(0	un^{+}
	$\langle^{+}c$	$(^{-}\infty$	$^{-}\langle ac\rangle$	[0]	(0	$^{+}\langle ac\rangle$	$^{-}\langle bc\rangle$	0)	[0]	$^{+}\langle bc\rangle$	$^{+}\infty)$
B_{Rd}	$^{-}d\rangle$	$^{+}\infty)$	$^{+}\langle ad\rangle$	[0]	0)	$^{-}\langle ad\rangle$	$^{+}\langle bd\rangle$	(0	[0]	$^{-}\langle bd\rangle$	$(^{-}\infty$
	0)	un^{+}	(0	[0]	0)	0)	(0	(0	[0]	0)	un^{-}
	0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]	[0]
	$^{+}d\rangle$	$(^{-}\infty$	$^{-}\langle ad\rangle$	[0]	(0	$^{+}\langle ad\rangle$	$^{-}\langle bd\rangle$	0)	[0]	$^{+}\langle bd\rangle$	$^{+}\infty)$
	$^{+}\infty)$	$(^{-}\infty$	$(^{-}\infty$	[0]	un^{+}	$^{+}\infty)$	$(^{-}\infty$	un^{-}	[0]	$^{+}\infty)$	$^{+}\infty)$

(a) Interval multiplication with ($^{-}\infty$, 0 and $^{+}\infty$).

		B_{La}					B_{Rb}				
	$\div$	$(^{-}\infty$	$\langle^{-}a$	[0	(0	$\langle^{+}a$	$^{-}b\rangle$	0)	0]	$^{+}b\rangle$	$^{+}\infty)$
B_{Lc}	$(^{-}\infty$	un^{+}	(0	[0]	0)	0)	(0	(0	[0]	0)	un^{-}
	$\langle^{-}c$	$^{+}\infty)$	$^{+}\langle a/c\rangle$	[0]	0)	$^{-}\langle a/c\rangle$	$^{+}\langle b/c\rangle$	(0	[0]	$^{-}\langle b/c\rangle$	$(^{-}\infty$
	[0	un	un	un	un	un	un	un	un	un	un
	(0	$(^{-}\infty$	$(^{-}\infty$	[0]	un^{+}	$^{+}\infty)$	$(^{-}\infty$	un^{-}	[0]	$^{+}\infty)$	$^{+}\infty)$
	$\langle^{+}c$	$(^{-}\infty$	$^{-}\langle a/c\rangle$	[0]	(0	$^{+}\langle a/c\rangle$	$^{-}\langle b/c\rangle$	0)	[0]	$^{+}\langle b/c\rangle$	$^{+}\infty)$
B_{Rd}	$^{-}d\rangle$	$^{+}\infty)$	$^{+}\langle a/d\rangle$	[0]	0)	$^{-}\langle ad\rangle$	$^{+}\langle b/d\rangle$	(0	[0]	$^{-}\langle b/d\rangle$	$(^{-}\infty$
	0)	$^{+}\infty)$	$^{+}\infty)$	[0]	un^{-}	$(^{-}\infty$	$^{+}\infty)$	un^{+}	[0]	$(^{-}\infty$	$(^{-}\infty$
	0]	un	un	un	un	un	un	un	un	un	un
	$^{+}d\rangle$	$(^{-}\infty$	$^{-}\langle a/d\rangle$	[0]	(0	$^{+}\langle a/d\rangle$	$^{-}\langle b/d\rangle$	0)	[0]	$^{+}\langle b/d\rangle$	$^{+}\infty)$
	$^{+}\infty)$	un^{-}	0)	[0]	(0	(0	0)	0)	[0]	(0	un^{+}

(b) Interval division with ($^{-}\infty$, 0 and $^{+}\infty$).

Figure 7.5: **Extended interval multiplication and division with ($^{-}\infty$, 0 and $^{+}\infty$):** The tables show the outcome for the multiplication and division operations including when one or both of the operands is an exclusive infinite bound or zero. The left angle bracket "⟨" denotes either "(" or "[". Similarly, the right angle bracket "⟩" denotes either ")" or "]". "[0]" denotes the bound is inclusive zero. "un^{+}" denotes the answer is undefined but positive; "un^{-}" denotes the answer is undefined but negative; "un" denotes the answer is undefined. Whether the result is a left or right bound is determined whenever it is chosen to be the minimum or maximum by the algorithms for multiplication and division.

can simply be discarded and that the correct answer is still available by choosing the minimum for the left bound and the maximum for the right bound.

Example 7.2.2. *Consider Figure 7.7. When we apply the formula for interval multiplication, in this case the product* **ac** *is* $(0 \times (^{-}\infty$ *which, reading from Figure 7.5(a), yields* un^{-} *, an undefined negative. Put this result to one side for the time being. Inspection of the remaining products shows that* **bc** *must be the minimum, since it is* $(^{-}\infty$ *. So now we seek the maximum, which must be a right bound. Now consider the products* **ad** *which yields the exclusive bound* $0)$ *and* **bd** *which yields the inclusive bound* $bd]$ *. Now, it must be the case that* $b > 0$ *since otherwise interval* $(0, b]$

$$[\overset{a}{24}\ ,\ \overset{b}{{}^{+}\infty})\ \text{x}\ [\overset{c}{2}\ ,\ \overset{d}{3}] = [48\ ,\ {}^{+}\infty)$$

	ac	*ad*	*bc*	*bd*
	[24 x [2	[24 x 3]	$^{+}\infty$) x [2	$^{+}\infty$) x 3]
Product	48	72	$^{+}\infty$	$^{+}\infty$
Limit	i	i	e	e
Min/Max	min		max	max

Figure 7.6: **Interval multiplication where one bound is an exclusive infinity**: The products are calculated having regard for the special cases listed in Figure 7.5(a). Algorithm 7.2.1 is then applied to decide if the bound is inclusive (i) or exclusive (e). The minimum is the left bound; the maximum is the right bound. Refer to **Example 7.2.1**.

$$(\overset{a}{0}\ ,\ \overset{b}{\text{b}}]\ \text{x}\ (\overset{c}{{}^{-}\infty}\ ,\ \overset{d}{\text{d}}] = ({}^{-}\infty\ ,\ B_R$$

if d < 0 then B_R = 0)
if d ≥ 0 then B_R = bd]

	ac	*ad*	*bc*	*bd*
	(0 x ($^{-}\infty$	(0 x d]	b] x ($^{-}\infty$	b] x d]
Product	un^{-}	0	$^{-}\infty$	bd
Limit	-	e	e	i
Min/Max	-	max if d < 0	min	max if d ≥ 0

Figure 7.7: **Interval multiplication where one product is undefined**: The undefined product may be discarded. The correct minimum and maximum occurs in the remaining products. Refer to **Example 7.2.2**.

would be null. Therefore:

- *if d* ≥ 0 *then the maximum is* bd]
- *if d* < 0 *then the maximum is* 0)

Finally, reconsider the undefined product **ac**. *The most we can say about this product is it must be negative. Then it cannot be greater than* 0) . *So in both cases, the maximum we choose for the right bound is correct.*

We give four further examples in Figure 7.8 (page 206). In each case, our objective is to demonstrate that we can safely discard the undefined products and choose our minimum (to form the left bound) and our maximum (to form the right bound) from the remaining products.

Example 7.2.3. *Consider Figure 7.8. In each case we can apply an analogous argument to the one advanced for Example 7.2.2 above. That is, whenever an undefined product is formed, we may safely discard it and choose the minimum and maximum from the remaining products that are defined. The final example in Figure 7.8(d) shows two products* **ac** *and* **bd** *yielding an undefined result. In both cases, the most we can say is that these products are negative. Therefore, they cannot be greater than* 0) *. But this is precisely the maximum already identified. Therefore,* 0) *is the correct maximum and forms the right bound of the resultant interval.*

a b c d

(0 , 2) x (5 , $^{+}\infty$) = (0 , $^{+}\infty$)

	ac	*ad*	*bc*	*bd*
	(0 x (5	(0 x $^{+}\infty$)	2) x (5	2) x $^{+}\infty$)
Product	0	un^{+}	10	$^{+}\infty$
Limit	e	-	e	e
Min/Max	min	-		max

(a)

a b c d

[-4 , 0) x ($^{-}\infty$, 1) = (-4 , $^{+}\infty$)

	ac	*ad*	*bc*	*bd*
	[-4 x ($^{-}\infty$	[-4 x 1)	0) x ($^{-}\infty$	0) x 1)
Product	$^{+}\infty$	-4	un^{+}	0
Limit	e	e	-	e
Min/Max	max	min	-	

(b)

a b c d

[-4 , 0) x [-2 , $^{+}\infty$) = ($^{-}\infty$, 8]

	ac	*ad*	*bc*	*bd*
	[-4 x [-2	[-4 x $^{+}\infty$)	0) x [-2	0) x $^{+}\infty$)
Product	8	$^{-}\infty$	0	un^{-}
Limit	i	e	e	-
Min/Max	max	min		-

(c)

a b c d

($^{-}\infty$, 0) x (0 , $^{+}\infty$) = ($^{-}\infty$, 0)

	ac	*ad*	*bc*	*bd*
	($^{-}\infty$ x (0	($^{-}\infty$ x $^{+}\infty$)	0) x (0	0) x $^{+}\infty$)
Product	un^{-}	$^{-}\infty$	0	un^{-}
Limit	-	e	e	-
Min/Max	-	min	max	-

(d)

Figure 7.8: **Extended interval multiplication where a product is undefined**: In each case, the undefined products may be discarded. The correct minimum and maximum are chosen from the remaining products to form the left and right bounds respectively of the resultant interval. Refer to **Example 7.2.3**.

7.2.5.1 Summary of Extended Interval Multiplication

We now summarise Extended Interval Multiplication (EIM), the innovative qualities of our proposed algorithm and how it differs from orthodox interval multiplication (IM). In the points that follow, the intervals we refer to are all numeric intervals over the Real numbers. We emphasize our algorithm applies to arithmetic with numeric *intervals*, not numeric arithmetic *per se*. Figure 7.5(a) (page 204), which exhibits a compelling symmetry, tabulates all the cases that occur in EIM for multiplication between two bounds.

- EIM allows multiplication between intervals that comprise both inclusive and exclusive bounds. IM defines multiplication for intervals comprising inclusive bounds only. EIM therefore allows for a richer set of cases.

- EIM calculates with operands that are *bounds*, not values; i.e., the *limit* is always included in the calculation. This allows us to distinguish between, for example, multiplying by $[0$ and multiplying by $(0$. We argue the different semantics of these two cases is clear. In the first case we are multiplying by zero which will always yield zero as the answer. In the second case we are multiplying by a positive real number that *approaches* zero.

- EIM disallows multiplication when one of the operands is one of the inclusive infinities: $^{+}\infty]$ or $[^{-}\infty$. The smallest numeric bound we allow for the purposes of arithmetic is $(^{-}\infty$; the largest is $^{+}\infty)$ [5]. This differs from IM which in general defines products with the infinities and an arbitrary operand as yielding an infinity of the appropriate sign (Hickey et al. 2001, Walster 2000, IEEE 1985).

- EIM allows multiplication when one or both operands *approaches* infinity; i.e., when the operands include the bounds $^{+}\infty)$ or $(^{-}\infty$. The values associated with these bounds are clearly Real numbers and therefore we argue that the meaning of products between these bounds and other Real bounds is well defined. For example, consider the product $^{+}\infty) \times [0$ which must yield an inclusive zero. The corresponding example from orthodox interval multiplication is $^{+}\infty] \times [0$ the result of which is undefined (Hickey et al. 2001).

- EIM defines multiplication in the case where one or both operands approaches zero; i.e., when the operands include the bounds $(0$ or $0)$. At the level of floating point implementation, these two bounds are analogous to the IEEE numeric constants 0^{+} and 0^{-} (IEEE 1985). However, continuing this analogy, the IEEE rules dictate that products between 0^{+} and 0^{-} and the two

[5]See Section 3.4.3 (page 52) for a precise definition of bound order.

IEEE infinities yield the result NaN - “not a number” (IEEE 1985). EIM gives a subtly different result which turns out to be pivotal to the correctness of the algorithm, namely that while we do not attempt to provide a numeric answer for such a product, we can nevertheless be sure of its sign. This gives rise to the notion of un^{+} (“undefined positive”) and un^{-} (“undefined negative”) which we introduced in Section 7.2.4.2 (page 201). This is turn allows us to safely discard products that are undefined when we pick the minimum and maximum bounds to form the left and right bounds of our answer. In fact, products between the exclusive zeros and the exclusive infinities are the only undefined cases in EIM.

7.3 Interval List Arithmetic

In this section we describe how interval arithmetic can be extended to arithmetic with interval lists. We firstly define what we mean by arithmetic with interval lists. We then focus on interval division and show how the need for an interval list structure arises as a natural consequence of the need to contain the answer for certain types of interval division.

7.3.1 Defining Arithmetic with Interval Lists

In this section we define what we mean by arithmetic with interval lists. We introduce this idea with an example involving interval addition.

Example 7.3.1. *Consider the interval list* $L = \{(1,2], [3,4)\}$. *Now consider the interval* $I = [5,6]$. *We wish to add this interval to each interval comprising L:*

$$\begin{aligned} L + I &= \{(1,2], [3,4)\} + [5,6] \\ &= \{(6,8], [8,10)\} \\ &= \{(6,10)\} \end{aligned}$$

We arrive at the last line because our definition of an interval list (Definition 3.8.3, page 71) dictates that the list is the logical disjunction of each interval and requires that all intervals are disjoint. Thus the two intervals that result from the addition: $(6,8]$ *and* $[8,10)$ *merge into a single interval*[6].

Algorithmically, arithmetic with an interval list as the first operand and an interval as the second operand seems straightforward. We simply carry out the operation on each interval comprising the list, forming the disjunction of the answers.

[6]See Section 3.8 (page 69) for a complete description of the interval list structure.

Definition 7.3.1. ***Interval list, interval arithmetic***: *Let $L = \sum_{i=1}^{n} I_i$ be an arbitrary numeric interval list comprising n disjoint intervals over the Real numbers (n = $0, 1, \cdots$)*[7]*. Let J be a numeric interval over the Real numbers. Let "$\otimes$" denote any of the four interval arithmetic operations: $\otimes \in \{+, -, \times, \div\}$. Let "$\vee$" denote the boolean "or" operator. Then $L \otimes J$ is defined as follows:*

$$
\begin{aligned}
L \otimes J &= \left(\sum_{i=1}^{n} I_i \right) \otimes \; J \\
&= (I_1 \vee I_2 \vee \cdots \vee I_n) \otimes J \\
&= (I_1 \otimes J) \vee (I_2 \otimes J) \vee \cdots \vee (I_n \otimes J) \\
&= \sum_{i=1}^{n} (I_i \otimes J)
\end{aligned}
$$

In the above definition, we see we are replacing each interval comprising interval list L with the result of applying the binary operator $\otimes$ with operand J. We might equally well consider arithmetic with the operands reversed; i.e., with an interval as the first operand and an interval list as the second operand. The following example illustrates this.

Example 7.3.2. *Consider the interval $I = [-1, 3]$ and an interval list $L = \{(1, 2], (2, 3)\}$. We wish to replace each interval comprising list L with the product formed when we multiply by I:*

$$
\begin{aligned}
I \times L &= [-1, 3] \;\times\; \{(1, 2], (3, 4)\} \\
&= \{[-1, 3] \times (1, 2] \;,\; [-1, 3] \times (3, 4)\} \\
&= \{[-2, 6], (-4, 12)\} \\
&= \{(-4, 12)\}
\end{aligned}
$$

We arrive at the last line because interval $(-4, 12)$ subsumes interval $[-2, 6]$ and our interval list must comprise disjoint intervals.

We now give a definition, analogous to Definition 7.3.1 above, for interval list arithmetic where an interval as the first operand and an interval list is the second operand.

Definition 7.3.2. ***Interval, interval list arithmetic***: *Let $L = \sum_{j=1}^{m} J_j$ be an arbitrary numeric interval list comprising m disjoint intervals over the Real numbers (m = $0, 1, \cdots$) Let I be a numeric interval over the Real numbers. Let "$\otimes$" denote any of the four interval arithmetic operations: $\otimes \in \{+, -, \times, \div\}$. Let "$\vee$" denote the*

[7] Throughout this chapter we employ the summation notation "$\sum$" to denote boolean disjunction. This was first introduced in Section 3.8.1 (page 70). Whenever this is expanded we will use "$\vee$" as the connector, rather than "+" to avoid confusion with numerical addition.

boolean "or" operator. Then $I \otimes L$ is defined as follows:

$$\begin{aligned} I \otimes L &= I \otimes \left(\sum_{j=1}^{m} J_i \right) \\ &= I \otimes (J_1 \vee J_2 \vee \cdots \vee J_m) \\ &= (I \otimes J_1) \vee (I \otimes J_2) \vee \cdots \vee (I \otimes J_m) \\ &= \sum_{j=1}^{m} \left(I \otimes J_j \right) \end{aligned}$$

We now combine Definitions 7.3.1 and 7.3.2 to arrive at a definition for arithmetic with two interval lists as the operands.

Definition 7.3.3. ***Interval list, interval list arithmetic****: Let $L_1 = \sum_{i=1}^{n} I_i$ and $L_2 = \sum_{j=1}^{m} J_j$ be two arbitrary numeric interval lists comprising disjoint intervals over the Real numbers. Let "$\otimes$" denote any of the four interval arithmetic operations: $\otimes \in \{+, -, \times, \div\}$. Let "$\vee$" denote the boolean "or" operator. Then we define $L_1 \otimes L_2$ in the following way:*

$$\begin{aligned} L_1 \otimes L_2 &= \left(\sum_{i=1}^{n} I_i \right) \otimes \left(\sum_{j=1}^{m} J_j \right) \\ &= (I_1 \vee I_2 \vee \cdots \vee I_n) \; \otimes \; (J_1 \vee J_2 \vee \cdots \vee J_m) \\ &= \quad (I_1 \vee I_2 \vee \cdots \vee I_n) \otimes J_1 \\ &\quad \vee (I_1 \vee I_2 \vee \cdots \vee I_n) \otimes J_2 \\ &\quad\quad \vdots \\ &\quad \vee (I_1 \vee I_2 \vee \cdots \vee I_n) \otimes J_m \\ &= \sum_{j=1}^{m} \sum_{i=1}^{n} \left(I_i \otimes J_j \right) \end{aligned}$$

Our primary motivation for defining interval list arithmetic is to provide a vehicle to extend interval division. This is described in detail in the following section.

7.3.2 Extending Interval Division

We now focus on interval division and show how the simple restrictions on interval arithmetic we set out above in Section 7.2.4 (page 198) allow us to derive sensible and expressive answers to an extended range of interval divisions.

We gave the orthodox formula for interval division in Section 7.2 (page 194) above as:

$$A \div B = [min\,(S_2)\,, max\,(S_2)]$$

where S_2 is the set of quotients $\{a/c, a/d, b/c, b/d\}$. Interval division is conventionally not defined when the interval divisor (B in the formula above) includes zero (Muñoz & Lester 2005, Hickey et al. 2001). However, we propose to treat these cases in a different way, by splitting interval divisors which include zero into two disjoint intervals. We begin with some examples of interval division where one of the operands approaches infinity or zero. This is followed by an example where the divisor includes zero.

Example 7.3.3. *Consider Figure 7.9 (page 212) which illustrates interval division, including some difficult cases where one of the operands approaches infinity or zero. In each case we refer to Figure 7.5 (page 204) to provide answers which are subject to the restrictions we impose. We apply the same Algorithm 7.2.2 (page 200) as for interval multiplication. That is, we apply the orthodox interval division formula to the values associated with each bound, having regard for the restrictions we impose in Section 7.2.4.2 (page 201). Then we apply Algorithm 7.2.1 (page 199) to decide if each bound is inclusive or exclusive. Finally, we choose the minimum and maximum, discarding bounds that are undefined, to form the left and right bounds respectively of the resultant interval.*

Consider Figure 7.9(d) in particular. The maximum bound in this case is clearly $+\infty)$ *which is given by quotient* $\mathbf{b}/\mathbf{c}$ *and this forms the right bound. The most that can be said of undefined quotient* $\mathbf{a}/\mathbf{c}$ *is that it is positive. So it cannot be less than* $(0$. *Therefore, it may be safely discarded since the remaining quotients both yield* $(0$ *, which forms the left bound.*

Consideration of Figure 7.9(e) leads to a similar argument. Again the undefined quotient may be safely discarded since in this case it cannot be greater than $0)$ *and this is precisely the bound yielded by quotient* $\mathbf{b}/\mathbf{d}$.

We now consider an example where the divisor includes the point zero.

Example 7.3.4. *Consider Figure 7.10 which depicts interval division with a divisor* $[-2, 1]$. *Conventionally, this division would be disallowed since the point zero is included. We propose to treat this case in a different way, by splitting the divisor into the two disjoint intervals* $[-2, 0)$ *and* $(0, 1]$. *We now proceed to operate on the dividend with the interval* list $\{[-2, 0), (0, 1]\}$. *Definition 7.3.2 tells us how to proceed. The answer is an interval list, rather than a single interval; i.e., the result of the division is the disjunction of the two intervals comprising the answer list.*

We argue that the result of the interval division in Example 7.3.4, expressed as an interval list, has clear semantics and is more expressive and useful than simply disallowing the division. We complete this section with some further examples of interval division where the divisor includes zero. In each case our objective is to

$$\overset{a}{[24}\ ,\ \overset{b}{{}^{+}\infty)} \div \overset{c}{[2}\ ,\ \overset{d}{3]} = [8\ ,\ {}^{+}\infty)$$

	a/c	*a/d*	*b/c*	*b/d*
	[24 / [2	[24 / 3]	$^{+}\infty$) / [2	$^{+}\infty$) / 3]
Quotient	12	8	$^{+}\infty$	$^{+}\infty$
Limit	i	i	e	e
Min/Max		min	max	max

(a)

$$\overset{a}{[24}\ ,\ \overset{b}{30)} \div \overset{c}{[2}\ ,\ \overset{d}{{}^{+}\infty)} = (0\ ,\ 15)$$

	a/c	*a/d*	*b/c*	*b/d*
	[24 / [2	[24 / $^{+}\infty$)	30) / [2	30) / $^{+}\infty$)
Quotient	12	0	15	0
Limit	i	e	e	e
Min/Max		min	max	min

(b)

$$\overset{a}{[24}\ ,\ \overset{b}{30)} \div \overset{c}{[-1}\ ,\ \overset{d}{0)} = ({}^{-}\infty\ ,\ -24]$$

	a/c	*a/d*	*b/c*	*b/d*
	[24 / [-1	[24 / 0)	30) / [-1	30) / 0)
Quotient	-24	$^{-}\infty$	-30	$^{-}\infty$
Limit	i	e	e	e
Min/Max	max	min		min

(c)

$$\overset{a}{(0}\ ,\ \overset{b}{10)} \div \overset{c}{(0}\ ,\ \overset{d}{{}^{+}\infty)} = (0\ ,\ {}^{+}\infty)$$

	a/c	*a/d*	*b/c*	*b/d*
	(0 / (0	(0 / $^{+}\infty$)	10) / (0	10) / $^{+}\infty$)
Quotient	un^{+}	0	$^{+}\infty$	0
Limit	-	e	e	e
Min/Max	-	min	max	min

(d)

$$\overset{a}{({}^{-}\infty}\ ,\ \overset{b}{-5]} \div \overset{c}{[5}\ ,\ \overset{d}{{}^{+}\infty)} = ({}^{-}\infty\ ,\ 0)$$

	a/c	*a/d*	*b/c*	*b/d*
	($^{-}\infty$ / [5	($^{-}\infty$ / $^{+}\infty$)	-5] / [5	-5] / $^{+}\infty$)
Quotient	$^{-}\infty$	un^{-}	-1	0
Limit	e	-	i	e
Min/Max	min	-		max

(e)

Figure 7.9: **Extended interval division**: We refer to Figure 7.5 to provide answers which are subject to the restrictions we impose on interval division. Algorithm 7.2.1 is then applied to decide if each bound is inclusive or exclusive. Any undefined divisions may be discarded. The correct minimum and maximum are chosen from the remaining answers to form the left and right bounds respectively of the resultant interval. Refer to **Example 7.3.3**.

demonstrate that the arithmetic we propose is simple to apply and yields answers which have a clear meaning.

Example 7.3.5. *Consider Figure 7.11 (page 214) which shows three further examples of interval division where the divisor includes zero. The examples of Figure 7.11(a) and Figure 7.11(b) together with the previous example of Figure 7.10*

$$\begin{aligned}[24, 30) \div [-2, 1] &= [24, 30) \div \{\ [-2, 0)\ ,\ (0, 1]\ \} \\ &= \{\ (^-\infty, -12]\ ,\ [24, {}^+\infty)\ \}\end{aligned}$$

	a/c	*a/d*	*b/c*	*b/d*
	[24 / [-2	[24 / 0)	30) / [-2	30) / 0)
Quotient	-12	$^-\infty$	-15	$^-\infty$
Limit	i	e	e	e
Min/Max	max	min		min
	[24 / (0	[24 / 1]	30) / (0	30) / 1]
Quotient	$^+\infty$	24	$^+\infty$	30
Limit	e	i	e	e
Min/Max	max	min	max	

Figure 7.10: **Interval division where the divisor includes the point zero**: The divisor is first split into two disjoint intervals such that the point $[0, 0]$ is excluded. We then apply the algorithm for division with interval lists. Refer to **Example 7.3.4**.

(page 213) show that when bounds comprising the dividend are of opposite sign and the divisor includes zero, the answer is always the entire set of Reals. This rule applies even when the bounds include the exclusive infinites (Figure 7.11(c)).

7.3.2.1 Summary of Extended Interval Division

We now summarise Extended Interval Division (EID), the innovative qualities of our proposed algorithm and how it differs from orthodox interval division (ID). In the points that follow, the intervals we refer to are all numeric intervals over the Real numbers. We emphasize our algorithm applies to arithmetic with numeric *intervals*, not numeric arithmetic per se. Figure 7.5(b), which exhibits a compelling symmetry, tabulates all the cases that occur in EID for division between two bounds.

- EID allows division between intervals that comprise both inclusive and exclusive bounds. ID defines division for intervals comprising inclusive bounds only. EID therefore allows for a richer set of cases.

- EID calculates with operands that are bounds, not values; i.e., the limit is always included in the calculation. This allows us to distinguish between, for example, dividing by $[0$ which is undefined and dividing by $(0$ which is defined.

- EID disallows division when one of the operands is one of the inclusive infinities, $^+\infty]$ or $[^-\infty$, but allows the calculation to proceed with the exclusive infinities, $^+\infty)$ or $(^-\infty$, yielding an analogous set of results to those typically defined for ID (Hickey et al. 2001, Walster 2000, IEEE 1985).

[-24 , 30) ÷ [-2 , 1] = [-24 , 30) ÷ { [-2 , 0) , (0 , 1] }
= { (${}^-\infty$, ${}^+\infty$) , (${}^-\infty$, ${}^+\infty$) }
= { (${}^-\infty$, ${}^+\infty$) }

	a/c	*a/d*	*b/c*	*b/d*
	[-24 / [-2	[-24 / 0)	30) / [-2	30) / 0)
Quotient	12	${}^+\infty$	-15	${}^-\infty$
Limit	i	e	e	e
Min/Max		max		min
	[-24 / (0	[-24 / 1]	30) / (0	30) / 1]
Quotient	${}^-\infty$	-24	${}^+\infty$	30
Limit	e	i	e	e
Min/Max	min		max	

(a)

[-10 , -2] ÷ (-2 , 1] = [-10 , -2] ÷ { (-2 , 0) , (0 , 1] }
= { (1, ${}^+\infty$) , (${}^-\infty$, -2) }

	a/c	*a/d*	*b/c*	*b/d*
	[-10 / (-2	[-10 / 0)	-2] / (-2	-2] / 0)
Quotient	5	${}^+\infty$	1	${}^+\infty$
Limit	e	e	e	e
Min/Max		max	min	max
	[-10 / (0	[-10 / 1]	-2] / (0	-2] / 1]
Quotient	${}^-\infty$	-10	${}^-\infty$	-2
Limit	e	i	e	i
Min/Max	min		min	max

(b)

(${}^-\infty$, 10) ÷ (-5, ${}^+\infty$) = (${}^-\infty$, 10) ÷ { (-5, 0) , (0, ${}^+\infty$) }
= { (${}^-\infty$, ${}^+\infty$) , (${}^-\infty$, ${}^+\infty$) }
= { (${}^-\infty$, ${}^+\infty$) }

	a/c	*a/d*	*b/c*	*b/d*
	(${}^-\infty$ / (-5	(${}^-\infty$ / 0)	10) / (-5	10) / 0)
Quotient	${}^+\infty$	${}^+\infty$	-2	${}^-\infty$
Limit	e	e	e	e
Min/Max	max	max		min
	(${}^-\infty$ / (0	(${}^-\infty$ / ${}^+\infty$)	10) / (0	10) / ${}^+\infty$)
Quotient	${}^-\infty$	un$^-$	${}^+\infty$	0
Limit	e	-	e	e
Min/Max	min	-	max	

(c)

Figure 7.11: **Further examples of extended interval division**: Whenever the divisor includes the point zero, we first split it into the two disjoint intervals on either side of zero. We then apply the algorithm for division with interval lists. We argue that the result of such interval division, expressed as an interval list, has clear semantics and is more expressive and useful than simply disallowing the division. Refer to **Example 7.3.5**.

- EID defines division in the case where the divisor approaches zero; i.e., when the divisor is one of the bounds (0 or 0) . The results are analogous to the IEEE floating point division with the numeric constants 0^+ and 0^- (IEEE 1985). However, the IEEE rules dictate that divisions where both dividend and divisor are 0^+ or 0^- and divisions where both dividend and divisor are one of the two IEEE infinities always yield the result *NaN* (“not a number”). EID gives a subtly different result which turns out to be pivotal to the correctness of the algorithm, namely that while we do not attempt to provide a numeric answer for such a division, we can nevertheless be sure of its sign. This gives rise to the notion of un^{+} ("undefined positive") and un^{-} ("undefined negative") which we introduced in Section 7.2.4.2 (page 201). This is turn allows us to safely discard divisions that are undefined when we pick the minimum and maximum bounds to form the left and right bounds of our answer.

- ID conventionally disallows division when the interval divisor includes the point zero (Muñoz & Lester 2005, Hickey et al. 2001). However, we argue that such cases can be handled by rewriting the divisor as two disjoint intervals on either side of the point zero. We utilise the interval list data structure first defined in Section 3.8 (page 69) to contain this rewritten divisor and then proceed to carry out the computation with an interval dividend and interval list divisor.

7.4 Temporal Intervals

In this section we examine temporal intervals where the intervals are conceived to specifically represent periods of time. We begin by describing how simple date arithmetic, as implemented in commercial RDBMS, can be easily extended to date intervals. This is followed by a brief description of how Allen's interval algebra (Allen 1983) can be implemented within our own interval algebra without modification. We then examine how Allen's interval algebra can be extended by our implementation.

One of our objectives in describing these extensions is to demonstrate the ease with which they can be implemented utilising the interval algebra we already have in place. Our implementation is sufficiently general to allow, for example, numeric interval arithmetic (Section 7.2, page 194), date interval arithmetic (Section 7.4.1, page 215) and Allen's interval algebra (Section 7.4.2, page 217) to be implemented with very little extension or modification to the functionality described in Chapter 3.

7.4.1 Extending Date Arithmetic to Intervals

Our interval implementation allows a numeric interval to be added to or subtracted from a date interval. We employ the same semantics as the Oracle RDBMS in this regard in that any *numeric* quantity added or subtracted from a date is interpreted as a quantity of days (Ashdown 2005*b*). For example[8], the following calculation adds

[8]In the date arithmetic examples that follow in this chapter, for clarity we simplify the syntax by omitting the quotes and format masks that would be required in the real RDBMS environment.

4 days to the date "23 June 2006":

```
23-JUNE-2006 + 4 = 27-JUNE-2006
```

This convention is easily extended by replacing the date in the above calculation with a date *interval* and the number with a numeric *interval*. It is helpful to consider temporal intervals as expressions of the form "`[StartTime,EndTime]`".

Example 7.4.1. *In this example we add the numeric interval `I = [-3,2]` to the date interval `D = [23-JUNE-2006,24-JUNE-2006]`.*

```
D + I = [23-JUNE-2006,24-JUNE-2006] + [-3,2]
      = [20-JUNE-2006,26-JUNE-2006]
```

Consider the semantics of Example 7.4.1 above. The resultant interval includes all the dates from three days before the the original start time until two days after the original end time. These semantics are clear and unambiguous. We must however be careful to note that we have overloaded the "+" operator because in the above example the operation requires one operand to be of type *date* and the other operand to be of type *number* and the numeric operand to be understood to refer to an interval of days.

We allow subtraction of date *intervals*. Again we employ the same semantics as the Oracle RDBMS: two dates can be subtracted to yield the difference in days between them (Ashdown 2005*b*). In the following example we employ intervals to express an uncertainty in the actual value of a date.

Example 7.4.2. *A conservation exercise to assure the survival of a rare bird records the hatch date of an individual to be between 01 June 2006 and 02 June 2006. Later the hatchling is found dead on 30 June 2006 and autopsy is only able to determine time of death to within 10 days. What was the age of the hatchling at death? Let `H` be the hatch date interval and `D` be the death date interval.*

```
    H = [01-JUNE-2006,02-JUNE-2006]
    D = [20-JUNE-2006,30-JUNE-2006]
D - H = [20-JUNE-2006,30-JUNE-2006] -
        [01-JUNE-2006,02-JUNE-2006]
      = [18,29]
```

The hatchling was between 18 and 29 days old at death.

In Example 7.4.2 above we have again overloaded the operator. Subtraction in this case means subtraction of date intervals, not subtraction of a numeric interval from a date. In our implementation of date interval arithmetic in the object-oriented Oracle PL/SQL environment, this overloading is performed transparently and the compiler infers the correct method to invoke from the data type of the parameters.

7.4.2 Allen's Interval Algebra

We now turn our attention to the interval algebra first proposed by Allen in (Allen 1983) and subsequently utilised by many researchers, for example (Gao, Jensen, Snodgrass & Soo 2005, Mani, Pustejovsky & Sundheim 2004, Krokhin, Jeavons & Jonsson 2003, Kriegel, Pötke & Seidl 2001, Nebel & Bürckert 1995, Özsoyoglu & Snodgrass 1995, Kim & Chakravarthy 1992, Maiocchi, Pernici & Barbic 1992). Allen's interval algebra is most often invoked in the context of *temporal intervals* and *temporal reasoning*. While temporal reasoning *per se* is beyond the scope of this thesis, we briefly focus on Allen's interval algebra because, as we have described in Chapter 3, our interval algebra was conceived to reason about *general* intervals. Our description is based around the three atomic data types: *numeric*, *string* and *date*, conventionally employed by RDBMS. In fact our algebra requires only that the data type concerned has a well defined total ordering[9]. We should therefore be well placed to represent Allen's temporal intervals within our own implementation.

We demonstrated above in Section 7.4.1 that our interval algebra is easily adapted to perform date interval arithmetic. The objective of this current section is to demonstrate that, similarly, we can easily incorporate Allen's interval algebra into our own interval algebra. We begin with a brief description of Allen's interval algebra and then show how the *13 basic interval relations* defined by Allen are directly implemented without modification by our own interval implementation. We then propose an extension to the 13 basic interval relations and suggest how this extension might be utilised.

7.4.2.1 The 13 Allen Interval Relations

Allen's interval algebra is based on the possible relationships between pairs of temporal intervals[10]. Allen intervals are always inclusive and of non-zero duration (Krokhin et al. 2003, Allen 1983)[11]. This observation leads to the following definition.

Definition 7.4.1. ***Allen Interval**: An Allen Interval, A, is a numeric interval with the following properties.*

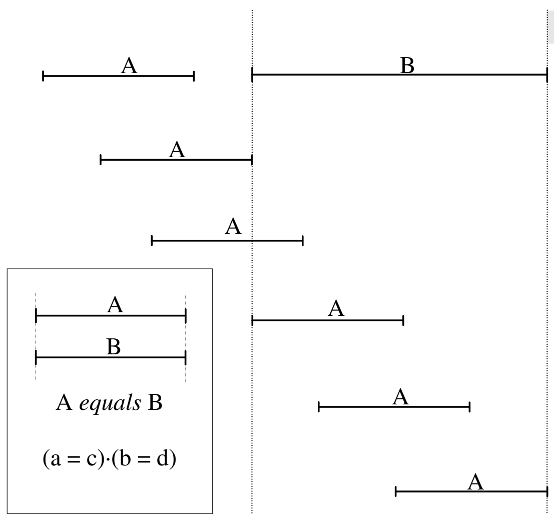
$$A = [a, b] \qquad \textit{where } a < b, \quad a, b \in R$$

[9]See Section 3.2 (page 45) for a precise description of the assumptions we invoke as axioms for the development of our interval algebra.

[10]For clarity, we will call these *Allen interval*s to distinguish them from our own interval representation which is more general than Allen's.

[11]We use the term "duration" to mean precisely the difference in time between the left and right endpoints of a temporal interval. If $A = [a, b]$ is an Allen interval, then its duration $d = b - a$. Allen intervals therefore require that $d > 0$.

Consider Figure 7.12. The 13 basic interval relations can be deduced from first principles by imagining an Allen interval A sliding from the left over another Allen interval B. Six of the relations have obvious inverses. For example, "*A before B*"



Relation	Function
A *before* B B *after* A	b < c
A *meets* B B *met-by* A	b = c
A *overlaps* B B *overlapped-by* A	(a < c)·(c < b)·(b < d)
A *starts* B B *started-by* A	(a > c)·(b < d)
A *during* B B *contains* A	(a = c)·(b < d)
A *finishes* B B *finished-by* A	(b = d)·(a > c)

Figure 7.12: **The 13 Allen interval relations**: Interval $A = [a, b]$. Interval $B = [c, d]$. The *Relation* column lists the first six basic relations and their inverses. The *Function* column expresses the relation as a boolean function of the endpoints $a, b, c, d \in R$. The symbol "·" denotes the boolean "and" operator. The "*equals*" relation, depicted on the left, brings the total to 13. The basic relations are all mutually exclusive; i.e., any two given Allen intervals are related by exactly one of the above basic relations.

immediately implies "*B after A*". These six plus their inverses supply 12 of the relations, plus the "*equals*" relation gives a total of 13 (Mani et al. 2004, Krokhin et al. 2003, Nebel & Bürckert 1995, Allen 1983).

The ability to *reason* about temporal intervals using the 13 basic relations is facilitated by considering the transitive relations that can exist between any three Allen intervals. Consider three arbitrary Allen intervals: A, B and C which are related in the following way:

$$A\ r_1\ B$$
$$B\ r_2\ C$$

where relations r_1 and r_2 are chosen from the 13 listed in Figure 7.12. Then the *transitive relations* that can exist between A and C are all the relations (chosen from the set of 13) that are logically possible, given r_1 and r_2. One of the main contributions of Allen's original paper (Allen 1983) was to tabulate all the $13 \times 13 = 169$ possible transitive relations that arise for "$(A\ r_1\ B) \cdot (B\ r_2\ C)$".[12] The following is a simple example of this type of inference and illustrates the usual sense of the term "transitivity".

[12]The symbol "·" denotes the boolean "and" operator.

Example 7.4.3. *Suppose intervals A and B are related by "A after B" and intervals B and C are related by "B after C". Then we may infer "A after C".*

The next example illustrates a more complex temporal inference[13].

Example 7.4.4. *Suppose "A overlaps B" and "B during C". Then we may write:*

$$(A\ overlaps\ B) \cdot (B\ during\ C) \Rightarrow (A\ overlaps\ C) \vee (A\ during\ C) \vee (A\ starts\ C)$$

7.4.2.2 Implementing The 13 Allen Interval Relations

We now show how we implement the 13 basic Allen interval relations using our own interval algebra. The objective of this section is not the details of the implementation, but to demonstrate that Allen's algebra is able to be directly implemented without modification. We are able to do this because our interval algebra is more general:

- We can represent intervals composed of bounds whose values are of type *numeric*, *string* or *date*. Therefore, temporal intervals can be represented using the *date* data type.
- We can represent intervals composed of bounds whose limits are inclusive or exclusive. Therefore, Allen intervals, which are always inclusive, can be represented.
- We can represent intervals that are null, represent a single point (i.e., a duration of zero), or have a positive duration. Therefore, Allen intervals, which must have a positive duration, can be represented.

Therefore, all that remains is to implement the 13 basic relations. These are embedded into the reasoning engine which forms the foundation of our semantic query optimizer.

7.4.2.3 Extending The 13 Allen Interval Relations

In this section we propose an extension to the 13 basic Allen interval relations and suggest how this extension might be utilised. Our proposed extension is based on the fact that intervals in our own algebra can be inclusive or exclusive whereas Allen intervals are always inclusive. Yet inspection of how the basic relations are defined in terms of their endpoints (the "*Function*" column in Figure 7.12, page 218) suggests the following extension:

[13]The symbol "$\vee$" denotes the boolean "or" operator. The symbol "$\Rightarrow$" denotes logical implication.

- Replace the endpoints "a, b, c, d" with their corresponding bounds "L_a, R_b, L_c, R_d" where "L_a, R_b" are the left and right bounds respectively of interval A and "L_c, R_d" are the left and right bounds respectively of interval B.
- Replace the boolean operators "$<, >, =$" with their overloaded versions which compare *bounds* (as opposed to comparing Real numbers), as defined in Section 3.4.3 (page 52).

The substitutions we suggest above are a continuation of the technique we have employed throughout this thesis; i.e., we manipulate and compare *bounds*, rather than values. The benefit that results in the case of the Allen relations is that the relations still hold but for both inclusive and exclusive intervals. Figure 7.13 sets out the basic Allen relations expressed first in their orthodox form in terms of the endpoints of the Allen intervals and second in their bound form where the bounds form the intervals, inclusive or exclusive, of our own algebra.

Allen Relation	Allen endpoint definition	Bound definition
A *before* B	$b < c$	$R_b < L_c$
A *meets* B	$b = c$	$R_b = L_c$
A *overlaps* B	$(a < c)\cdot(c < b)\cdot(b < d)$	$(L_a < L_c)\cdot(L_c < R_b)\cdot(R_b < R_d)$
A *starts* B	$(a = c)\cdot(b < d)$	$(L_a = L_c)\cdot(R_b < R_d)$
A *during* B	$(a > c)\cdot(b < d)$	$(L_a > L_c)\cdot(R_b < R_d)$
A *finishes* B	$(b = d)\cdot(a > c)$	$(R_b = R_d)\cdot(L_a > L_c)$
A *equals* B	$(a = c)\cdot(b = d)$	$(L_a = L_c)\cdot(R_b = R_d)$

Figure 7.13: **Extending the 13 Allen interval relations**: $A = \langle a, b\rangle$ is the interval consisting of left and right bounds L_a, R_b where $L_a = \langle a$ and $R_b = b\rangle$. $B = \langle c, d\rangle$ is the interval consisting of left and right bounds L_c, R_d where $L_c = \langle c$ and $R_d = d\rangle$. The symbol "$\langle$" denotes either "(" or "[". The symbol "$\rangle$" denotes either ")" or "]". The symbol "$\cdot$" denotes the boolean "and" operator. We replace each endpoint "a, b, c, d" in the orthodox Allen relation with the corresponding bound "L_a, R_b, L_c, R_d". The operators "$<, >, =$" are overloaded such that in the "*Allen endpoint definition*" column they compare Real numbers, whereas in the "*Bound definition*" column they compare bounds. The Allen relations still hold in this new formulation but now can be applied to both inclusive and exclusive intervals.

We complete this section by suggesting how our extended Allen interval relations might be utilised. The application we have in mind is inspired by the observation that *time* (and therefore *temporal* intervals) is represented in a variety of ways which often have subtly different semantics (Mani et al. 2004, Özsoyoglu & Snodgrass 1995, Snodgrass & Ahn 1985). For example, the fact that Allen intervals are required to have a non-zero duration and are always inclusive, strongly suggests the endpoints represent the “ticks” of some arbitrary clock, a *timestamp*, which is inherently limited to a precision of one tick. The following example concerns time representation in the Oracle RDBMS and illustrates why the ability to represent time within an exclusive interval allows a more accurate representation of time.

Example 7.4.5. *The Oracle RDBMS implementation of the* `date` *data type allows a precision of 1 second. However, a* `timestamp` *data type is also implemented which records system time*[14] *with a precision of up to* 10^{-9} *seconds (Agrawal 2005). Times may be readily converted between the two data types. For example, when converting from* `timestamp` *to* `date`*, the fractional seconds part is truncated. When converting from* `date` *to* `timestamp`*, the fractional seconds part is set to zero.*

Consider an application where the duration of a significant event is recorded by noting `START_TIME` *and* `END_TIME`*. Initially these times are recorded with data type* `date`*. Later it is decided to upgrade the precision and use data type* `timestamp`*. Suppose a particular tuple records* D_1 *and* D_2 *respectively as the start and end times in the old* `date` *data type. However, in the new* `timestamp` *data type, the most accurate way of representing this information is to include the uncertainties:*

D_1 *becomes the interval* $I_1 = \left[D_1, D_1'\right)$ *where* $D_1' = D_1 + 1s$

D_2 *becomes the interval* $I_2 = \left[D_2, D_2'\right)$ *where* $D_2' = D_2 + 1s$

Therefore the duration of this particular event is given by applying the extended interval subtraction formula from Section 7.2.3:

$$\begin{aligned} \textit{Duration} &= I_2 - I_1 \\ &= \left[D_2, D_2'\right) - \left[D_1, D_1'\right) \\ &= \left(D_2 - D_1',\ D_2' - D_1\right) \end{aligned}$$

7.5 Summary

In this chapter we highlight the versatility of the interval algebra we have implemented and how it can be utilised in a variety of areas with very little extension or modification. We focus firstly on *interval arithmetic* and show how the four basic arithmetic operations with intervals can be extended using our own interval algebra. We then focus on *temporal intervals* and show how the interval algebra of

[14]We use the term “system time” to mean the time as measured by the internal clock of the processor.

Allen (Allen 1983) can be extended using our own interval algebra. The main objective of this chapter is to highlight the versatility of the interval algebra we have implemented, that it is more general than the orthodox interval algebras of interval arithmetic (Hickey et al. 2001) and temporal intervals (Krokhin et al. 2003) and how it can be utilised in a variety of areas with very little extension or modification.

We now summarise the main contributions of this chapter.

- We begin by highlighting the principle differences in our interval algebra compared to orthodox treatments of intervals, namely our treatment of the two infinities and the treatment of limits (Section 7.2.1).

- We show how interval addition (Section 7.2.2) and interval subtraction (Section 7.2.3) can be generalised to include both inclusive and exclusive intervals while retaining sound semantics.

- We give a general algorithm for Extended Interval Arithmetic (EIA) (Section 7.2.4) showing first how inclusive and exclusive limits can be processed separately (Section 7.2.4.1) and then describing how the orthodox formulae for interval arithmetic can be extended to encompass both inclusive and exclusive bounds and the two exclusive infinities (Section 7.2.4.1). We then tabulate and review all the restrictions we apply to EIA and show how EIA allows a wider range of intervals to be treated compared with orthodox IA (Section 7.2.4.2).

- We give a detailed description of how we propose to extend interval multiplication (Section 7.2.5), highlighting the different way we treat the two infinities and how our restrictions introduce a subtly different semantics which turns out to be pivotal to the correct functioning of the algorithm for Extended Interval Multiplication. We tabulate the full range of cases for which the Extended Interval Multiplication is valid (Figure 7.5).

- We show how interval arithmetic is easily extended to arithmetic with interval *lists*, as we have defined them (Section 7.3.1) and sketch by way of examples how this extension might be utilised.

- We give a detailed description of how we propose to extend interval division (Section 7.3.2), highlighting the different way we treat the two infinities and how our restrictions introduce a subtly different semantics which turns out to be pivotal to the correct functioning of the algorithm for Extended Interval Division. We tabulate the full range of cases for which the Extended Interval Division is valid (Figure 7.5). We utilise our new definitions of arithmetic with interval lists to show how we propose to treat interval division where the divisor includes the point *zero*.

- We examine the special case of temporal intervals. We first show how *date arithmetic*, as it is typically defined in RDBMS, can be easily extended to arithmetic with date intervals (Section 7.4.1).
- We focus on the seminal work of Allen (Section 7.4.2) and show how Allen's interval algebra is completely expressible within our own interval algebra (Section 7.4.2.2). We then show how the 13 basic Allen interval relations can be extended by our own interval algebra while retaining their validity (Section 7.4.2.3).

Chapter 8

Conclusion And Future Work

8.1 Review

The primary aim of this thesis is to answer the question “Is semantic query optimization worthwhile?” in the context of relational database management systems.

To address this question we have firstly summarised the conclusions of other researchers in this field, describing the main types of semantic query optimization proposed so far and identifying the sources of semantic information which can be used to recast an SQL query into another form which can be answered more efficiently, while producing the same answer as the original query. *Semantic* query optimization is distinctly different from the optimization carried out by conventional SQL language optimizers, which ultimately rely on the rules of relational algebra to *syntactically* rewrite an SQL query such that it can be executed with near-optimal efficiency. Semantic information can be harvested from

- the schema meta-data (such as table and view definitions);
- constraints stored and maintained by the DBMS (such as *check* constraints);
- human domain experts;
- discovered rules identifying relationships between tabular data.

With regard to schema constraints stored and maintained by the DBMS, while these may be used to constrain data at insert and update time, other than a small proportion, they are ignored at query time by all current commercial SQL language optimizers.

The fact that current SQL language optimizers largely ignore semantic information is not due to an inherent failure or inefficiency but due to the nature of their design. These optimizers are primarily syntactic and they do not in general take even simple semantic information into account. For example, even if a particular column is declared within its table definition to be “`not null`”, a query against this table which asks for null values in this column will still be submitted to the database, invoking all the normal database activity, even though the answer set must logically be empty.

All semantic optimizers require a *reasoning engine* to *deduce* conclusions using premises which incorporate the semantic information harvested from the schema under consideration. We introduce an *interval algebra* which we use in a novel way and which forms the basis of our reasoning engine. The interval algebra is built using a small number of well established axioms; namely, we accept the *Boolean Algebra* and the existence of a deterministic *total ordering* for the data types we employ. We define our basic data structure, the *interval list*, as a set of disjoint intervals. We show that all the reasoning functionality we require can be built using our interval algebra to operate on interval lists in conjunction with our basic axioms. The main result we draw from our step by step theoretical development is the ability to perform a form of *conjunction*, *disjunction* and *negation* using interval lists. These results form the foundation of our reasoning engine implementation.

We describe in detail the design of a practical semantic query optimizer. Our semantic optimizer sits in front of the normal SQL optimizer and pre-processes the SQL queries before passing them to the normal SQL optimizer. An important feature of our design is that it employs meta-data already held as part of the RDBMS but which is typically only utilised to a very limited extent for the purposes of query optimization by current SQL optimzers. For example, our semantic optimizer harvests the various schema constraints such as *check*, *primary key* and *foreign key* constraints. We argue for the collection of a *query profile*, which is a high level description of what tables are actually queried, plus the columns that are actually cited in the restriction and join clauses. This knowledge can then be used to tightly focus a more extensive knowledge discovery exercise, thus avoiding the exponentially increasing expense of performing an exhaustive search for relationships within tabular data. To some extent, activity of this sort is already part of normal Database Administrator duties and leads, for example, to the creation of auxiliary data structures such as indexes and clusters which increase query efficiency. It is straightforward to collect such a query profile with existing commercial RDBMS.

We reiterate the conclusion of other researchers that the detection of unsatisfiable queries can form an important part of semantic query optimization. This is because unsatisfiable queries need not be submitted to the database at all, potentially saving the usual computational costs associated with such a query. Recognising the potential value of detecting unsatisfiable queries, we describe a simple but highly effective algorithm for enhancing the semantic information that leads to the detection of unsatisfiability. We show how the detection of “data holes” can proceed across all relevant dimensions (i.e., across all columns that are actually cited in query restrictions and join clauses) without impacting on database usability. This information is then incorporated into existing meta-data, increasing the probability that unsatisfiable queries will be detected before the query is actually submitted.

We highlight an inherent limitation in the effectiveness of much of the methodology of SQO. This limitation arises naturally from the fact that SQO depends in part on the existence of *anomalous* queries. For example, unsatisfiable queries or “out of range” queries in general might arise because of an incomplete or inaccurate knowledge of schema semantics. But if anomalous queries are never (or hardly ever) submitted, perhaps the effort of semantically optimizing queries is not worthwhile. On the other hand, the potential impact of a naïve user query on database usability might make the effort of semantically optimizing all queries worthwhile.

It might also be the case that queries are automatically generated, for example, by a GUI based tool which generates queries in response to a non-technical user's "point and click" actions. Therefore, SQO might be seen as a valuable technique in any situation where queries may not reflect an accurate knowledge of the actual schema semantics.

An important part of this thesis is its empirical component. We set out to discover if the efficiencies claimed for SQO would be confirmed in practice. To this end, we highlight the difficulties of obtaining consistent, repeatable results in RDBMS where automatic maintenance processes may execute at times outside of the control of the experimenter and where large query and data caches are available. Our goal is realism. We explain why it can be misleading to use total elapsed time only as the true measure of query cost. Instead we argue for a combined metric which incorporates the three metrics: *elapsed time*, *disk i/o* and *CPU time*. The optimizations we report have two crucial properties:

1. they are the *ratio* of optimized to unoptimized cost;

2. they are the *average* cost for batches consisting of many similar queries.

The first property serves to minimize the random experimental error and to minimise the number of variables we need to consider. The second property allows us to infer the likely efficiency gain from a whole class of similar queries, rather than individual, manually optimized queries.

Very few researchers in SQO report empirical results for queries against tables which are realistically sized and indexed. Our experiments are designed around tables which approximate conditions found in actual data warehouses. Crucially, our target tables are sensibly indexed. This is important because it is most unlikely in practice that tables of the size and nature we query in our experiments would not be indexed with normal B-tree indexes. We report results for both queries and equi-joins.

With regard to unsatisfiable queries and joins, taken as a whole our empirical results strongly support the hypothesis that detecting unsatisfiable queries is worthwhile. However, we show unequivocally, both with our cost model and our empirical results, that detection of unsatisfiable queries is not costless. Our cost model foreshadows and our empirical results confirm that there is an upper bound to the amount of optimization we can expect from detecting unsatisfiable queries and that the cost of detecting such queries can rapidly become comparable to and exceed the normal cost of processing the SQL query. This preprocessing cost is approximately four times higher for equi-joins than for queries and increases with increasing query difficulty. The key factor required to "break even" in this context is a sufficiently high probability of an unsatisfiable query occurring. In the case of queries, our prototype optimizer manages to break even with probabilities of approximately 5% to 10%, across a wide range of table sizes and query difficulty. In the case of equi-joins, our prototype optimizer manages to break even with probabilities of approximately 10% to 20%, across a wide range of table sizes and query difficulty.

Our empirical results also report the effect of removing two key phrases when they are redundant:

- Removing the "`distinct`" from "`select distinct`" when it can be deduced *a priori* that all rows returned will be distinct.

- Removing the restriction "`COL is not null`" when it can be deduced *a priori* that "`COL`" cannot be null.

These disarmingly simple textual changes can give rise to dramatically different execution costs, as *predicted* by the SQL optimizer. However, our results strongly suggest that the actual efficiency gain that results from removing these redundancies is, when averaged out over many queries, significantly less than what is suggested by the SQL optimizer's prediction. In the case of "distinct" removal, we obtained a useful 20% and 60% efficiency gain for queries and joins respectively. However, in the case of "not null" removal, we obtained only insignificant efficiency gains across a broad range of query difficulties and table sizes.

An important part of SQO identified by all researchers is the discovery of semantic rules[1] which relate tabular data in some way such that extra restrictions can be inferred (*restriction introduction*) or redundant restrictions removed (*restriction removal*). In the context of relational databases, a reasonable heuristic is to look for correlations between indexed and unindexed columns. For example, if a query restriction cites an unindexed column we might look for a rule which allows us to infer a restriction on an indexed column and introduce this extra restriction to the query. The objective of restriction introduction is to efficiently reduce the cardinality of the result set. The objective of restriction removal is to eliminate the redundant filtering of the result set.

The interval algebra we develop to form the basis of our reasoning engine is very general. Its innovative features include the following:

- Intervals may be both inclusive or exclusive.

- The four limits we use "(,) , [,]" are conceived to be *operators* which operate on *values* to produce left and right *bounds*.

[1]In this thesis we do not consider in detail the problems of actually discovering such rules.

- The values we enclose in our intervals may be any data type, provided only that the data type has a *deterministic total ordering.* For example, we may have *numeric intervals*, *string intervals* and *date intervals.*

We use the generality of our interval algebra to extend the four operations of *interval arithmetic.* We show that the subtly different semantics introduced by allowing both inclusive and exclusive intervals over the Real numbers allows *minus infinity* and *plus infinity* to be represented and incorporated meaningfully into arithmetic calculations with intervals. We further show how our extensions can be used to calculate with a wider set of cases which, for example, include division by intervals that include the point zero. We show it is straightforward to extend interval arithmetic to arithmetic with interval *lists*; i.e., sets of disjoint intervals.

We again highlight the versatility of our interval algebra by showing that it subsumes *Allen's interval algebra.* This interval algebra is conceived to operate specifically with *temporal intervals.* We show that the Allen algebra is a special case of our own interval algebra where we restrict the values to the temporal domain and where all limits are inclusive (i.e., we use only the limits "[,]"). However, when the restriction to inclusive limits is relaxed, we show the wider semantics that result are meaningful and useful for modeling certain temporal scenarios which cannot be captured by the Allen algebra.

We complete this review by returning to the central question "Is semantic query optimization worthwhile"?

- With regard to the detection of unsatisfiable queries, which features prominently in the research into SQO, we have shown the effectiveness depends on the probability of unsatisfiable queries actually occurring. If this probability is vanishingly small then other factors must be considered such as the impact on database usability of a naïve user query. Detection of unsatisfiable queries is not costless. However, if this cost is comparable to the computational costs incurred by the SQL optimizer, then we argue this optimization *is* worthwhile.

- With regard to the removal of redundant phrases from SQL query text, we have presented strong evidence that these simple textual changes can have an important positive impact. Viewing the question the other way around, there seems little reason *not* to implement these textual changes, provided the redundancy can be detected with a cost comparable to the computational costs already incurred by the SQL optimizer.

- With regard to restriction introduction and removal, our empirical results indicate restriction *removal* is likely to be the more successful strategy. This form of SQO is facilitated by the discovery of rules which correlate a highly selective indexed column with an unselective non-indexed column. In the case of restriction introduction, optimization was worthwhile only for queries returning a very small percentage of total table rows. In the case of restriction removal, optimization was worthwhile for a much wider range of query cardinalities. In general, our results suggest searching for rules which allow query restrictions on unselective columns to be eliminated.

8.2 Contributions

We now list the main contributions of this thesis.

- We present a thorough analysis of research in SQO. We introduce definitions that clarify and simplify the terminology used by other researchers. In addition, further definitions are introduced that enable a more detailed discussion (Chapter 2).

- We develop a sound theoretical base for our study using an *interval algebra* which we show may be built using only a small number of well understood and researched axioms. We extend the interval algebra by defining an *interval list* data structure which we subsequently utilize as the basic data structure of our implementation. To our knowledge, this is the first report of an interval algebra used in the way we describe and generalised to operate with any data type that has a deterministic total ordering (Chapter 3).

- We show how a practical semantic query optimizer may be built utilising readily available semantic information, much of it already captured by metadata typically stored in commercial RDBMS. We describe how SQO may proceed as a series of pre-processing steps which may be switched in and out as changing database conditions make different forms of SQO worthwhile. While other researchers have suggested the basic techniques we describe, we focus on the fact that certain types of SQO, such as the detection of unsatisfiable queries, are likely to be worthwhile *given a particular query profile*. We describe an extension to the detection of unsatisfiable queries which enables "data holes" to be discovered separately across all relevant dimensions (i.e., across all table columns that are *actually* cited in query restrictions) and incorporated incrementally into the semantic information utilised by the semantic optimizer with little or no impact on database usability. In addition, we develop a cost model which accurately predicts the amount of optimization we can expect and which sets a clear upper bound to this optimization. To our knowledge, this is the first report to explicitly highlight an inherent limitation on the effectiveness of detecting unsatisfiable queries and joins (Chapter 4).

- We describe an empirical methodology which overcomes problems of repeatability and consistency which typically arise in experiments with RDBMS where automatic maintenance processes may be invoked outside of the control of the experimenter and where large query and data caches are available. We do not report results for individual queries but instead report statistical *averages* that arise from large batches of similar queries. Our results therefore inform us as to what we can expect from whole classes of queries rather than individual queries specific to particular databases (Chapter 5).

- We present a series of empirical results arising from experiments to confirm the effectiveness or otherwise of various types of SQO. Our experiments are performed with tables which realistically reflect the conditions likely to be encountered in data warehouses. Crucially, we report results for tables that are realistically indexed. To our knowledge, this is the first report of empirical results for queries and equi-joins against tables that are indexed in this way and where the results are a statistical average for batches of many similar queries (Chapter 6).

- We describe several important extensions which utilise the interval algebra we describe in Chapter 3. Firstly, we show how our interval algebra can be used to implement a novel type of *interval arithmetic*. Our interval arithmetic is more general than traditional implementations in that we allow both inclusive and exclusive upper and lower bounds for the numeric intervals. Furthermore, we show how the subtly different semantics of our implementation elegantly capture notions such as plus and minus infinity while allowing arithmetic calculation to proceed across a greater set of cases than allowed for by traditional interval arithmetic. Secondly, we show how our interval algebra subsumes the temporal algebra of Allen (Allen 1983) and how the *13 Allen interval relations* can be meaningfully extended (Chapter 7).

8.3 Future Work

We now briefly list some of the future research which this thesis anticipates.

- Currently our semantic reasoning engine operates as a preprocessor sitting as a separate module in front of the normal SQL language optimizer. It is implemented in PL/SQL which is incorporated into the Oracle RDBMS. However, this is a software layer above the level at which SQL language optimization occurs[2]. We speculate the efficiency of the semantic optimizer could be im-

[2]The kernel of the Oracle RDBMS is implemented in C, as is the SQL language optimizer.

proved by a more intimate association with the SQL language optimizer. One impediment to this is that the Oracle SQL optimizer (in common with other commercial RDBMS) is not available for public scrutiny. However, other comparable "open source" RDBMS such as *MySQL*[3] and *PostgreSQL*[4] do publish the source code of their optimizers, making this a viable avenue for investigation.

- One feature of the algorithms we develop in Chapter 3 is that there are clear opportunities for parallelism. We speculate that a parallel implementation of the algorithms for conjunction and disjunction of interval lists would result in a significant speed up. For conjunctive queries in particular, the semantic preprocessing of all restrictions in parallel could result in a dramatic "short circuiting" in comparison to sequential processing, if just one of the restrictions is unsatisfiable.

- In Section 4.2, we describe how much of semantic query optimization, by its very nature, is limited in its effectiveness by the fact that it depends on the detection of queries which are, in some sense, anomalous. This raises the question as to how frequently anomalous queries occur *in practice*. We have not attempted to answer this question in this thesis. Empirical studies of a selection of real world database applications would provide quantitative data to address this question.

- In Section 4.4.2, we describe how we collect information about data holes which we subsequently utilise in our semantic optimizer. Although we explain how we constrain the complexity of the search for data holes, we make the assumption that the information is discovered off line and does not affect the efficiency of the semantic optimizer at run time. While this is a reasonable simplifying assumption, a quantitative analysis of data hole discovery would enhance the practical application of SQO.

- In this thesis we have focussed on databases with static schemas where data updates occur infrequently. This specifically excludes transactional databases where data updates typically occur frequently, possibly concurrently via multiple users. We pointed out in Chapter 2 that when data updates do occur, these might invalidate any rules that have been discovered through the analysis of data. However, if re-validation of this type of semantic rule can be accomplished in a time comparable to the mean period between queries, it could be practical to apply the techniques of SQO to transactional databases.

[3]See `http://www.mysql.com`
[4]See `http://www.postgresql.org`

- The implementation of interval arithmetic described in Chapter 7 is at an early stage. We wish to investigate its usability in particular with regard to the incorporation of techniques to deal with calculation with the two infinities and values that approach zero.

- Currently we are able to reason about intervals of type *numeric*, *string* or *date*. However, *any* data type which may be deterministically ordered can be implemented. The main practical requirement is a suitable "*compare*" function which unambiguously ranks the data type[5]. For example, the potential exists to reason about complex data types such as might be found in object-oriented databases where the same semantic optimization techniques we describe in this thesis might be applied.

- We have not studied *temporal databases* in this thesis. However, the subsumption of the *Allen algebra* which we describe in Chapter 7 leads naturally to the consideration of how the extended semantics of our proposed temporal algebra might facilitate reasoning in *temporal databases*.

[5]See Section 3.2

Appendix A

Supporting Empirical Results

A.1 Introduction

This Appendix presents detailed results for all experiments described in Chapter 6 “Empirical Results” for Sections 6.3 to 6.9. In Chapter 6, for clarity we included in the main text only summary results. This Appendix supplements these results with the experimental outcomes that lead to those summaries. Each section in this Appendix covers one complete experiment.

We do not report absolute cost metrics. Rather, we report the *ratio* of the *optimized* versus *unoptimized* cost metric. We judge the cost of a query by using three different cost metrics. These metrics are described in detail in Section 5.2.4. For clarity, Table 5.1 from Chapter 5 describing these metrics is repeated here in Table A.1.

Metric	**Meaning**	$\mathbf{R_{cost}} = \frac{\mathbf{COST_{opt}}}{\mathbf{COST_{norm}}}$
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement.	R_{cpu}
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.	R_{elpsd}
DISK	Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.	R_{dsk}
COMBINED	The average of the other three metrics. This metric is only ever reported as a *ratio*.	R_{com}

Table A.1: **Query cost metrics and their meaning**.

Two levels of detail are presented within each section.

- In the first subsection we present one full set of results showing the relative costs of optimized versus unoptimized queries for the combined cost metric ratio R_{com}. This includes a sample set of individual result graphs that led to the construction of the summary results, which are also displayed here. Refer to Figure A.1. The X-axis (independent variable) is the probability of an *unsatisfiable query* occurring in a given batch. The Y-axis (dependent variable) is the cost metric ratio; i.e., the ratio of the optimized cost versus the unoptimized cost. For example, R_{com} which denotes the *combined cost metric ratio* (see Table 5.1). This category of graph always includes the following features.

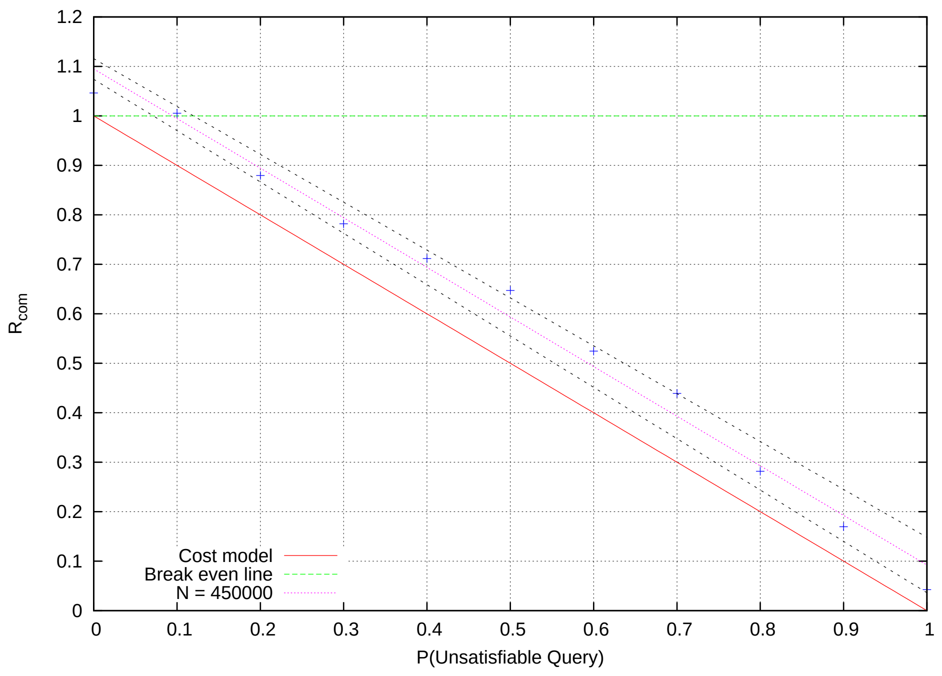


Figure A.1: **Typical two dimensional result graph**: The X-axis (independent variable) is probability of an *unsatisfiable query* occurring in a given batch of queries. The Y-axis (dependent variable) is the cost metric ratio; i.e., the ratio of the optimized cost versus the unoptimized cost.

– Experimental data points, marked as *blue* crosses.

– Least Squares Regression Line: The best linear fit for the experimental data points is computed and plotted as a *pink* line. We calculate the *regression line* for each result set using an implementation of the nonlinear least-squares (NLLS) *Marquardt-Levenberg algorithm* (Press et al. 1992) as implemented by *Gnuplot* (Broeker et al. 2006, Drakos & Moore 2006)[1].

– Standard Error Lines: Two standard error lines (plotted as a *black* dotted line) appear with the regression line to provide an indication of experimental uncertainty. These errors are typically known as “asymptotic standard errors” and represent the standard deviation of each parameter (Press et al. 1992).

– Idealised Cost Model Line: This is the graph of the function “$f(x) = 1 - x$” and represents the line predicted by the cost model developed

[1]*Gnuplot* is a portable command-line driven interactive data and function plotting utility. See `http://www.gnuplot.info`.

in Section 5.4 *where the time taken by the extra semantic optimizing step is negligible*. Therefore, in the absence of any other optimization, we cannot reasonably expect the cost metric ratio to be below this line. This is plotted as a *red* line.

  - Break Even Line: This line marks a cost metric ratio of 1, representing equal costs for both optimized and normal batches. Therefore, any results below this line represent a *positive* optimization; i.e., the semantically optimized cost is less than the normal cost. Conversely, any results above this line represent a *negative* optimization; i.e., the semantically optimized cost is actually more than the normal cost. This is plotted as a *green* line.

- In the second subsection we present summary results across all three cost metric ratios defined in Table A.1. We omit individual plots for these individual metric ratios. The summary graphs are all presented as three dimensional projections of the dependent variable versus the two independent variables. We calculate a *regression surface* for each result set using an implementation of the nonlinear least-squares (NLLS) *Marquardt-Levenberg algorithm* (Press et al. 1992) as implemented by *Gnuplot* (Drakos & Moore 2006). The form of the regression surface is given by the following:

$$f(x,y) = A + Bx + Cx^2 + Dy + Ey^2$$

  where $f(x,y)$ is the dependent variable, x and y are the independent variables, A,B,C,D,E are constants determined by the regression analysis.

A.2 Unsatisfiable Queries – No Indexing

This section contains a full analysis of the results reported in Section 6.3. The objective of these experiments is to establish a baseline with regard to the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size*. It is most unlikely in practice that tables of the size and makeup we query in our experiments would *not* be indexed. However, we are motivated to query a set of unindexed tables:

- to set a baseline against which our other experiments with tables that *are* realistically indexed may be compared;

- to relate our work with other published research that typically cite results for unindexed tables (Gryz et al. 1999, Cheng et al. 1999).

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Relative table size, denoted by the number of table rows $Rows$.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. Each query consists of a single restriction defined by one interval. None of the columns cited in query restrictions is indexed.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and $Rows$
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size $Rows$) disappears

A.2.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.2 plots R_{com} vs Probability of Unsatisfiable Query P and shows how consistently this ratio varies with increasing table size across four orders of magnitude. The results conform closely to the cost model for table rows $Rows = 1,000$ to $500,000$. Figure A.2(h) combines all results into a single graph and clearly shows the optimization achieved by recognising the unsatisfiable queries is independent of table size.

Figure A.3(a) plots R_{com} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ and summarises the results as a R_{com} surface. Figure A.3(b) shows the same surface along with the regression surface. Figure A.3(c) compares the idealised regression, "cost model" and the "break even" surfaces. Figure A.3(d) is exactly the same projection, but viewed by looking directly into the XZ plane. This clearly shows the R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted, with positive optimization occurring when $P > 5\%$. We have positive optimization across four orders of magnitude of table size.

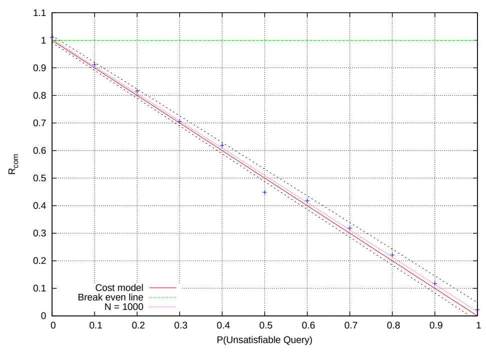
A.2.2 Individual Cost Metric Ratios

The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.2.

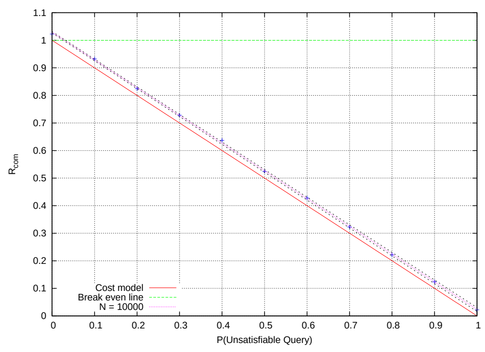
Figure	**Results Presented**
A.4	R_{cpu} vs Probability of Unsatisfiable Query P vs Rows N
A.5	R_{dsk} vs Probability of Unsatisfiable Query P vs Rows N
A.6	R_{elpsd} vs Probability of Unsatisfiable Query P vs Rows N

Table A.2: **Location of summary results** for the three individual cost metric ratios displaying R_{cost} vs Probability of Unsatisfiable Query P vs Rows N .

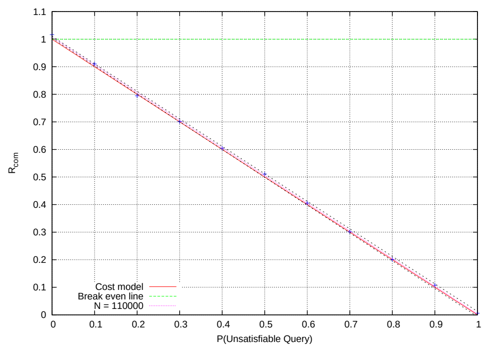
The individual cost metric ratios show more variation than the combined cost metric ratio R_{com} results displayed in Section A.2.1. This is described in detail in Chapter 5. We note however that the results show a consistent efficiency gain for the semantically optimized queries and a close correspondence to the results predicted from our cost model.



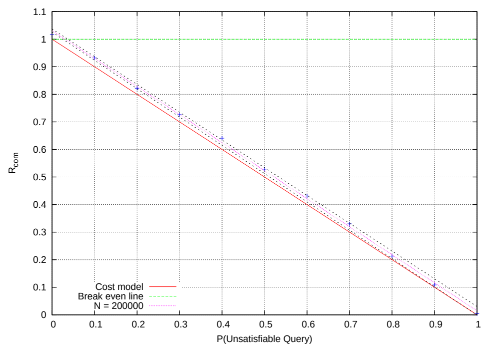
(a) Table rows `N = 1,000`



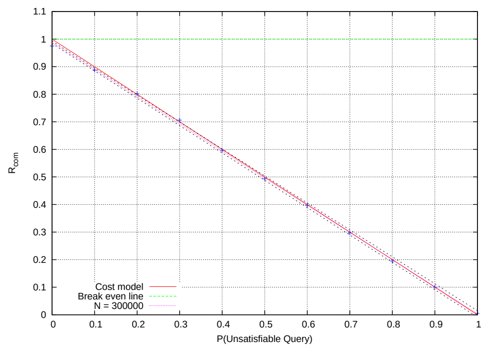
(b) Table rows `N = 10,000`



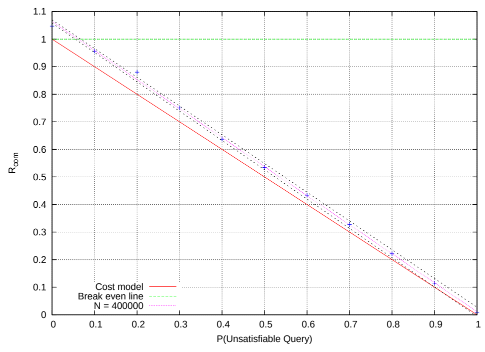
(c) Table rows `N = 110,000`



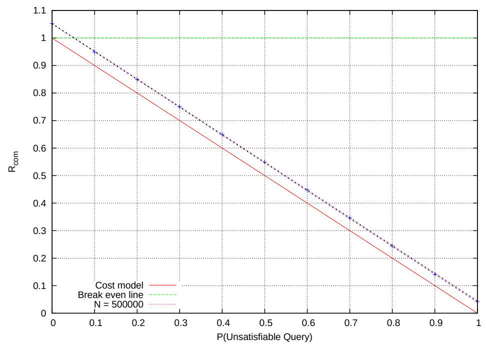
(d) Table rows `N = 200,000`



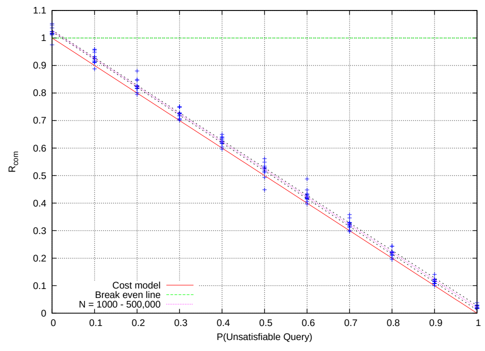
(e) Table rows `N = 300,000`



(f) Table rows `N = 400,000`

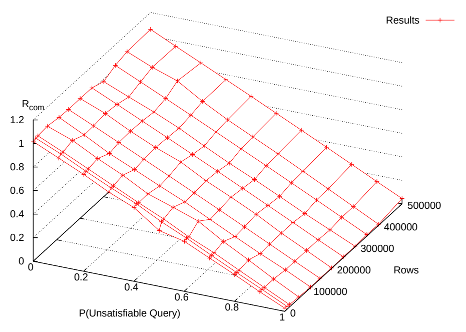


(g) Table rows `N = 500,000`

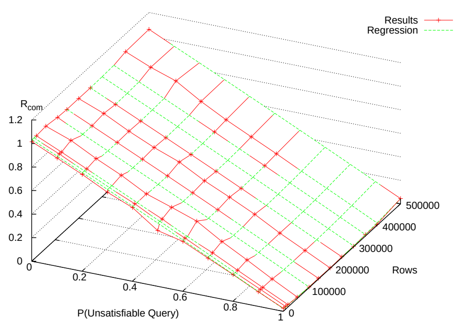


(h) Table rows `N = 1,000` to `500,000`

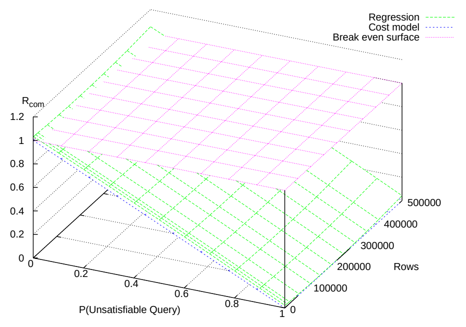
Figure A.2: **Ratio R_{com} vs Probability of Unsatisfiable Query P (no indexing)**: Figures A.2(a) to A.2(h) show how consistent ratio R_{com} is as table size increases from $Rows = 100$ to $500,000$. The results conform closely to the cost model. Figure A.2(h) combines all results into a single graph. The *combined* ratio R_{com} is the average of the other three cost metric ratios which we interpret as the overall query cost.



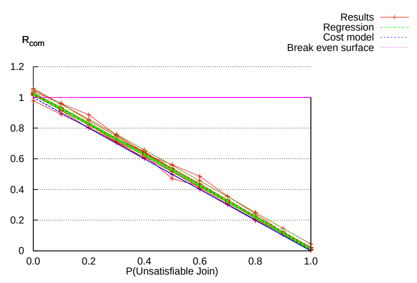
(a) R_{com} surface for `100` to `500,000` rows



(b) R_{com} surface with regression surface.

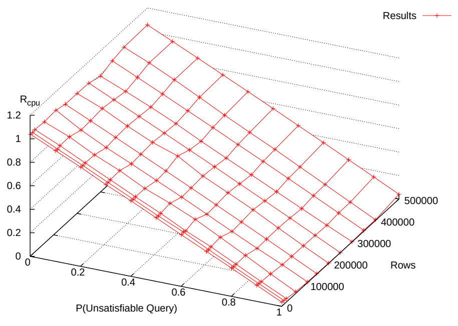


(c) Regression, "cost model" and "break even"surfaces.

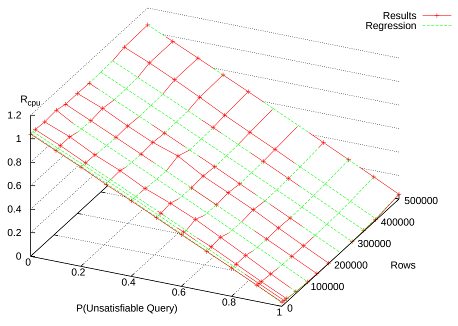


(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

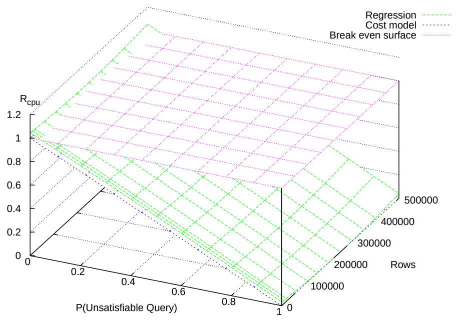
Figure A.3: **Ratio** R_{com} **vs Probability of Unsatisfiable Query** P **vs Relative Table Size** *Rows* **(no indexing)**: These figures summarise the results presented above in Figure A.2 as a R_{com} surface. Figures A.3(b) and A.3(c) show R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size, with the optimization cost rising slightly as table size becomes very large ($Rows > 400,000$).



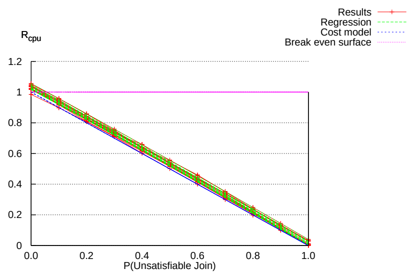
(a) R_{cpu} surface for `100` to `500,000` rows



(b) R_{cpu} surface with regression surface.

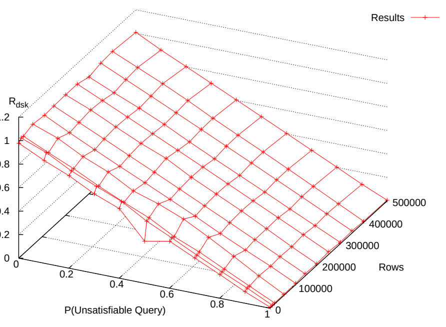


(c) Regression, "cost model" and "break even"surfaces.

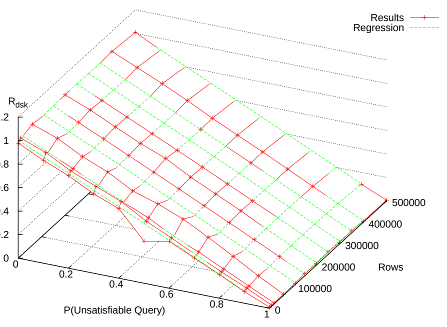


(d) R_{cpu}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

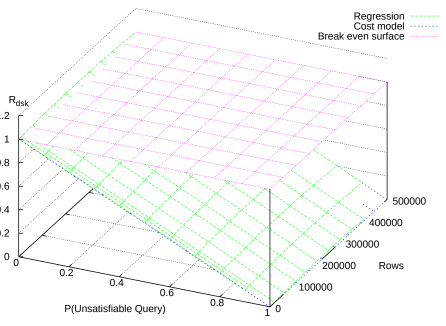
Figure A.4: **Ratio R_{cpu} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (no indexing)**: The R_{cpu} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. Figure A.4(c) provides compelling visual confirmation that R_{cpu} scarcely rises above 1 indicating we have positive optimization across four orders of magnitude of table size.



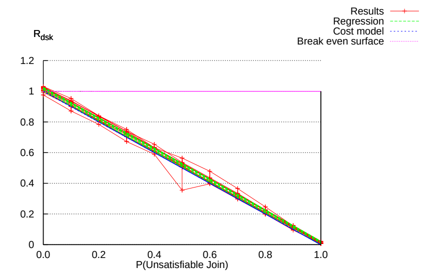
(a) R_{dsk} surface for `100` to `500,000` rows



(b) R_{dsk} surface with regression surface.

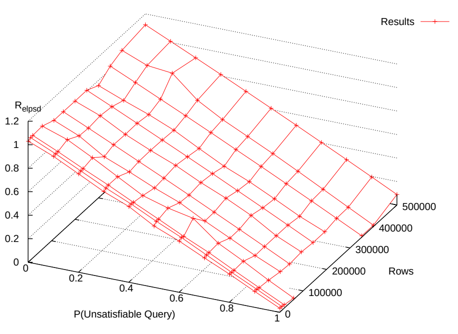


(c) Regression, “cost model” and “break even”surfaces.

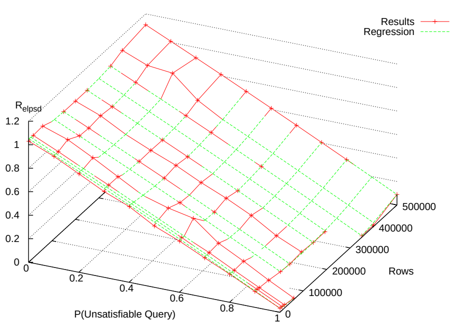


(d) R_{dsk}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

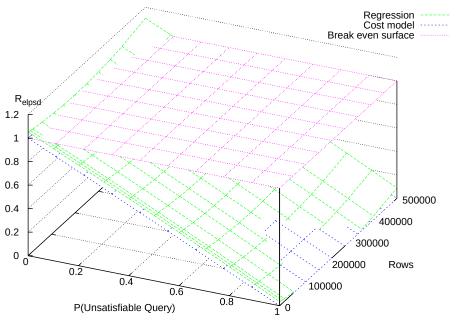
Figure A.5: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (no indexing)**: The R_{dsk} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. Figure A.5(c) provides compelling visual confirmation that R_{dsk} scarcely rises above 1 indicating we have positive optimization across four orders of magnitude of table size.



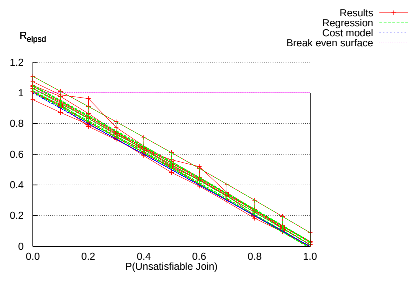
(a) R_{elpsd} surface for `100` to `500,000` rows



(b) R_{elpsd} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{elpsd}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure A.6: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (no indexing)**: The R_{elpsd} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. Figures A.6(c) and A.6(d) suggest optimization is degraded slightly with respect to elapsed time for very large table sizes ($Rows > 400,000$).

A.3 Indexed Unsatisfiable Queries

This section contains a full analysis of the results reported in Section 6.4. The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and *relative table size.* The methodology is identical to the experiments reported above in Appendix A.2, with the key difference that all columns cited in query restrictions are indexed with a "normal" B-tree index (Chan 2005*a*).

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Relative table size, denoted by the number of table rows $Rows$.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. Each query consists of a single restriction defined by one interval.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and $Rows$
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size $Rows$) disappears

A.3.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.7 plots R_{com} vs Probability of Unsatisfiable Query P and shows how consistently this ratio varies with increasing table size across four orders of magnitude. The results conform closely to the cost model for table rows $Rows = 100$ to $1,000,000$.

Figure A.8(a) plots R_{com} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ and summarises the results as a R_{com} surface. Figure A.8(b) shows the same surface along with the regression surface. Figure A.8(c) compares the idealised regression, “cost model” and the “break even” surfaces. Figure A.8(d) is exactly the same projection, but viewed by looking directly into the XZ plane. This clearly shows the R_{com} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted, with positive optimization occurring when $P > 10\%$, across four orders of magnitude of table size.

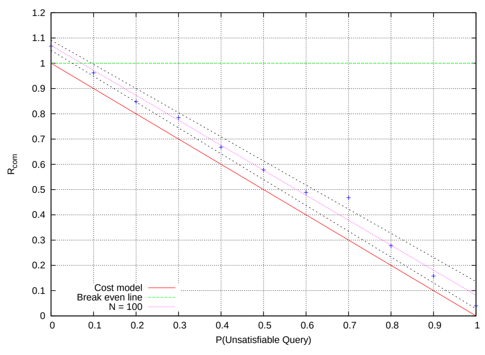
A.3.2 Individual Cost Metric Ratios

The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.3.

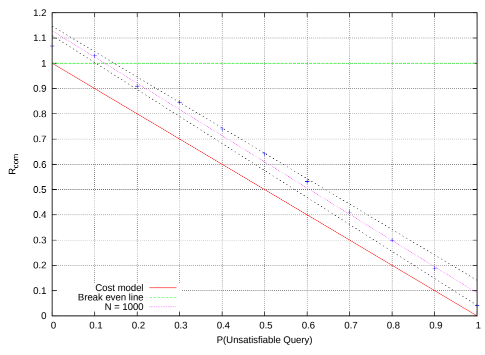
Figure	**Results Presented**
A.9	R_{cpu} vs Probability of Unsatisfiable Query P vs Rows N
A.10	R_{dsk} vs Probability of Unsatisfiable Query P vs Rows N
A.11	R_{elpsd} vs Probability of Unsatisfiable Query P vs Rows N

Table A.3: **Location of summary results** for the three individual cost metric ratios displaying R_{cnt} vs Probability of Unsatisfiable Query P vs Rows N .

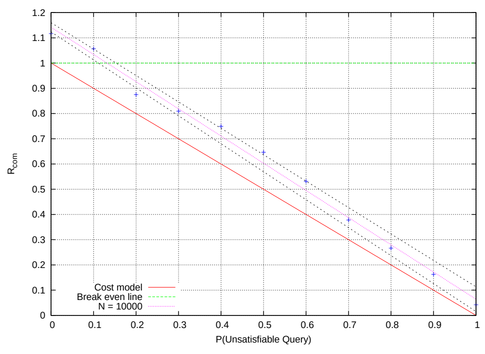
The individual cost metric ratios show more variation than the combined cost metric ratio R_{com} results displayed in Section A.3.1. This is described in detail in Chapter 5. We note however that the results show a consistent efficiency gain for the semantically optimized queries and a close correspondence to the results predicted from our cost model.



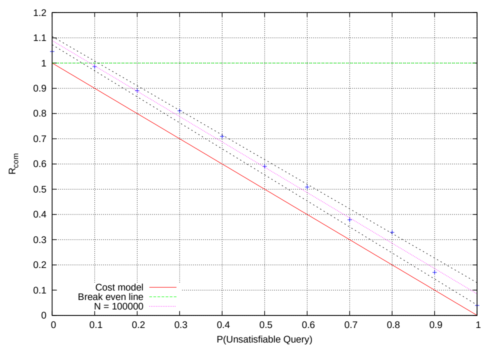
(a) Table rows `N = 100`



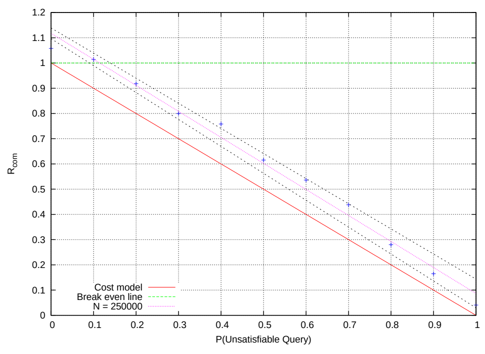
(b) Table rows `N = 1,000`



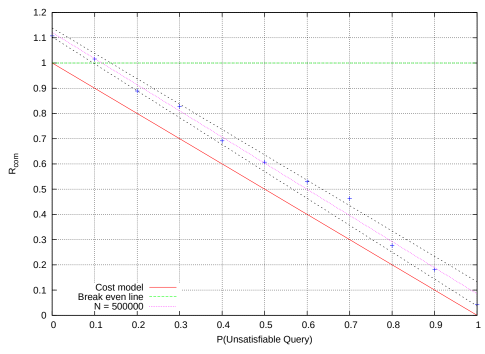
(c) Table rows `N = 10,000`



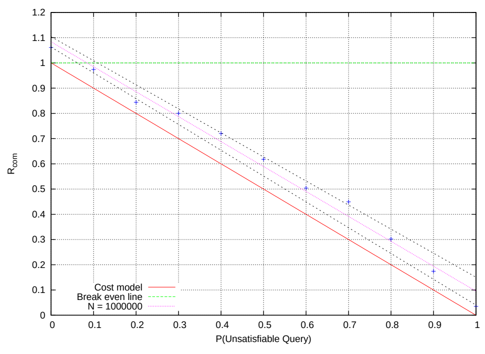
(d) Table rows `N = 100,000`



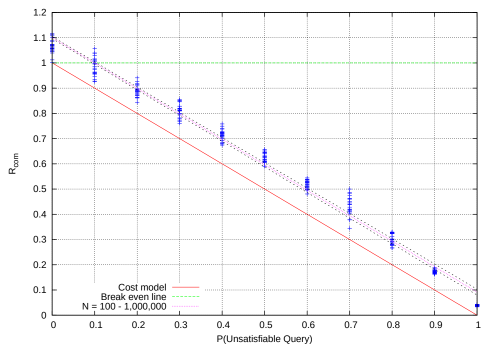
(e) Table rows `N = 250,000`



(f) Table rows `N = 500,000`

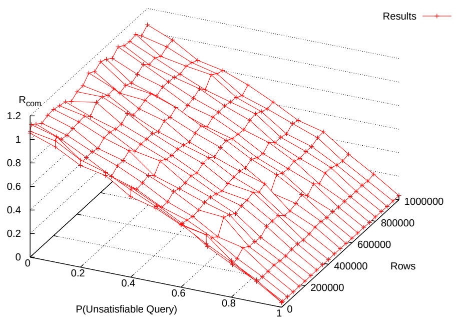


(g) Table rows `N = 1,000,000`

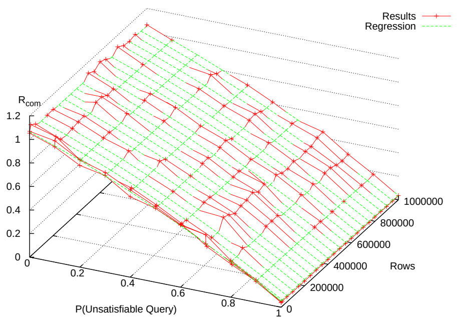


(h) Table rows `N = 100` to `1,000,000`

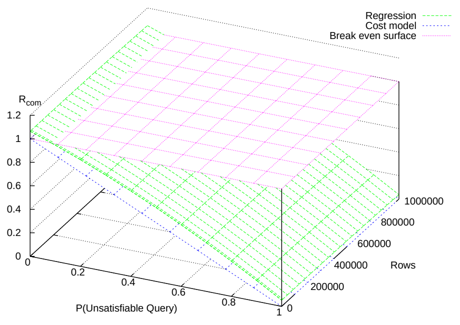
Figure A.7: **Ratio R_{com} vs Probability of Unsatisfiable Query P (indexed)**: Figures A.7(a) to A.7(h) show how consistently this ratio varies with increasing table size across four orders of magnitude. The results conform closely to the cost model for table rows $Rows = 100$ to $1,000,000$. Figure A.7(h) combines all results into a single graph. The *combined* ratio R_{com} is the average of the other three cost metric ratios which we interpret as the overall query cost.



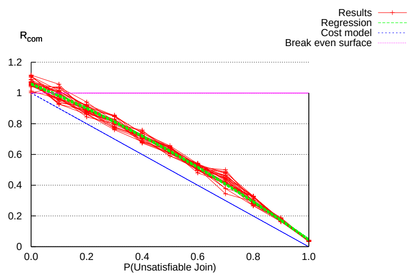
(a) R_{com} surface for `100` to `1,000,000` rows



(b) R_{com} surface with regression surface.

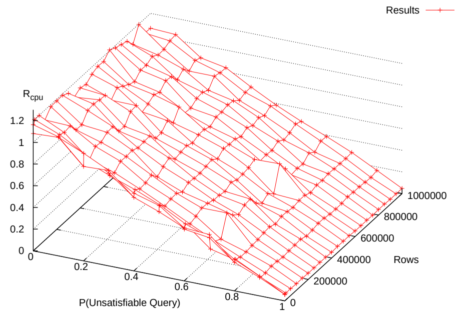


(c) Regression, “cost model” and “break even”surfaces.

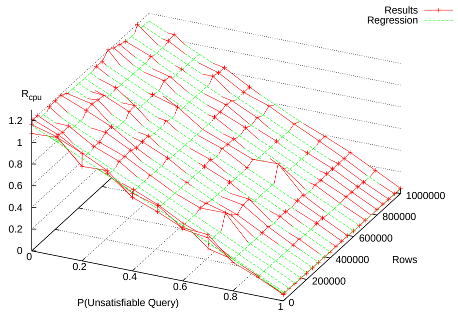


(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

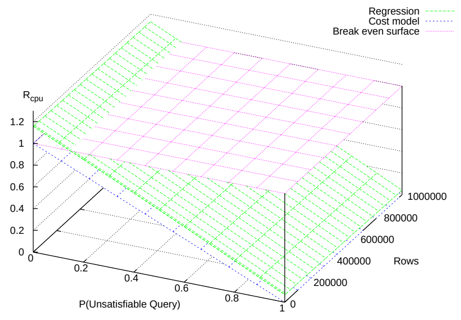
Figure A.8: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Relative Table Size *Rows* (indexed)**: These figures summarise the results presented above in Figure A.7 as a R_{com} surface. Figures A.8(b) and A.8(c) show the R_{com} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.1$.



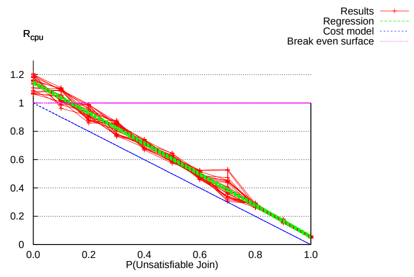
(a) R_{cpu} surface for `100` to `1,000,000` rows



(b) R_{cpu} surface with regression surface.

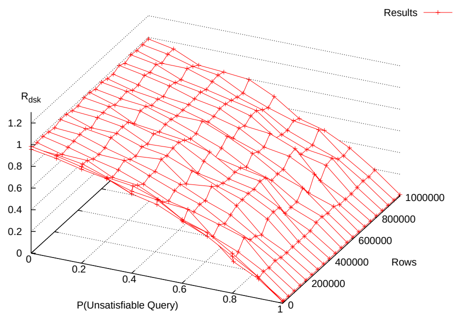


(c) Regression, "cost model" and "break even"surfaces.

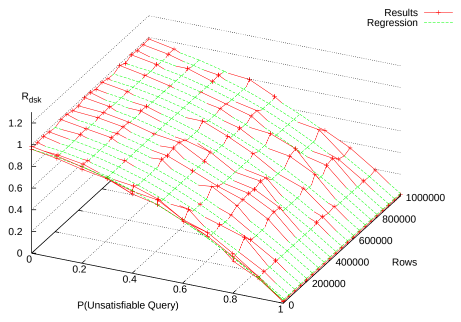


(d) R_{cpu}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

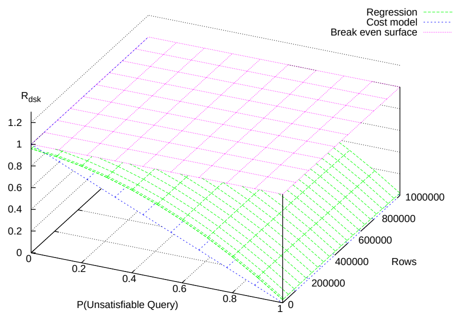
Figure A.9: **Ratio** R_{cpu} **vs Probability of Unsatisfiable Query** P **vs Relative Table Size** *Rows* **(indexed)**: These figures summarise the results presented above in Figure A.7 as a R_{cpu} surface. Figures A.9(b) and A.9(c) show R_{cpu} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.15$.



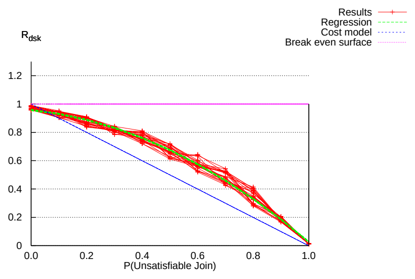
(a) R_{dsk} surface for `100` to `1,000,000` rows



(b) R_{dsk} surface with regression surface.

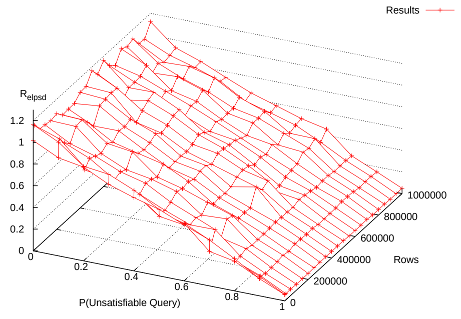


(c) Regression, “cost model” and “break even”surfaces.

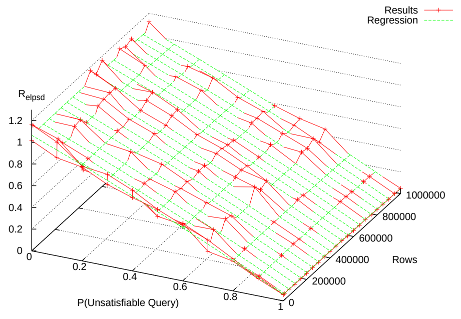


(d) R_{dsk}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

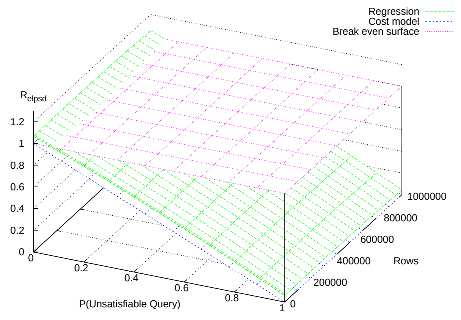
Figure A.10: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (indexed)**: The R_{dsk} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. Figure A.10(c) provides compelling visual confirmation that R_{dsk} scarcely rises above 1 indicating we have positive optimization across four orders of magnitude of table size.



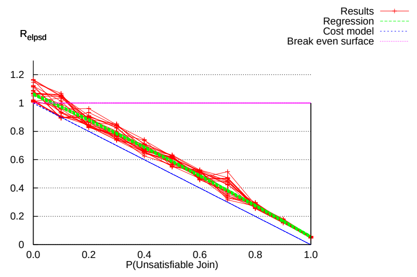
(a) R_{elpsd} surface for `100` to `1,000,000` rows



(b) R_{elpsd} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{elpsd}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure A.11: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Relative Table Size $Rows$ (indexed)**: These figures summarise the results presented above in Figure A.7 as a R_{elpsd} surface. Figures A.11(b) and A.11(c) show R_{elpsd} surface sits just above the "cost model surface", indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.05$.

A.4 Indexed Unsatisfiable Queries – Varying Restrictions per Query

This section contains a full analysis of the results reported in Section 6.5. The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and the *number of restrictions per query*. In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Number of restrictions per query R/Q. Each restriction is defined by a single interval.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in query restrictions are indexed with a "normal" B-tree index.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and R/Q
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (restrictions per query R/Q) disappears

A.4.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.12 plots R_{com} vs Probability of Unsatisfiable Query P and shows how this ratio varies with increasing numbers of restrictions per query (R/Q). Each restriction is defined by a single interval. The results show R_{com} increases as R/Q increases. This is what we expect since the cost of semantically pre-processing the query rises with increasing query complexity.

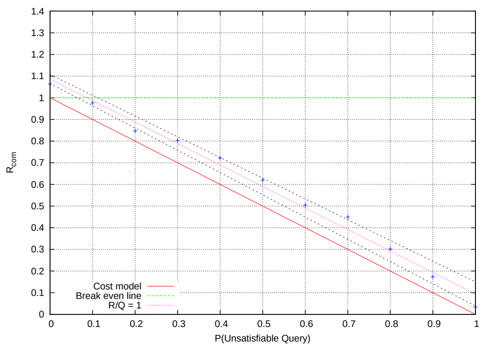
Figure A.13 plots R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q and summarises the results as a R_{com} surface. For low R/Q, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the “cost model surface”. However as R/Q rises, the pre-processing cost becomes significant and we require a greater proportion of unsatisfiable queries to make semantic optimization worthwhile.

A.4.2 Individual Cost Metric Ratios

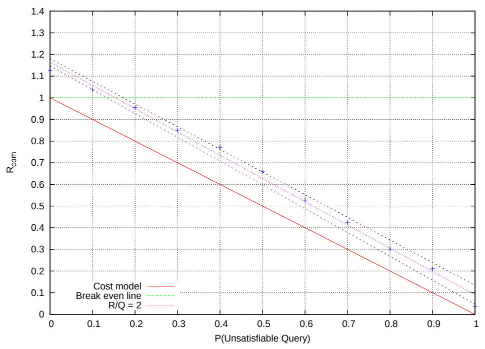
The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.4.

Figure	Results Presented
A.14	R_{cpu} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q
A.15	R_{dsk} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q
A.16	R_{elpsd} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q

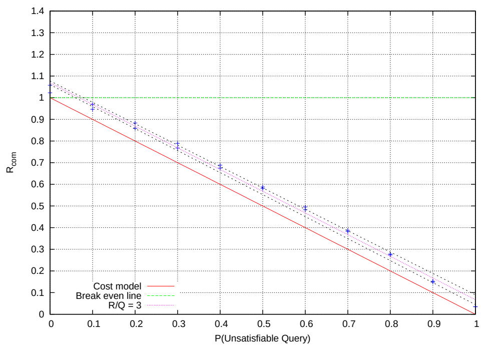
Table A.4: **Location of summary results** for the three individual cost metric ratios displaying R_{cost} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q.



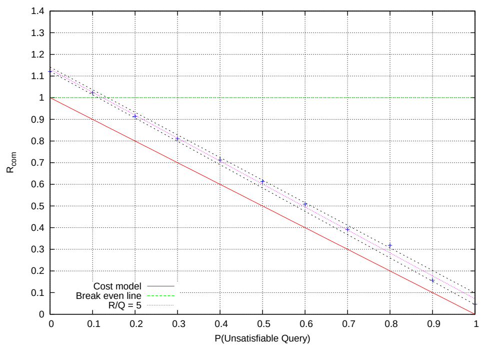
(a) Restrictions per Query $R/Q = 1$



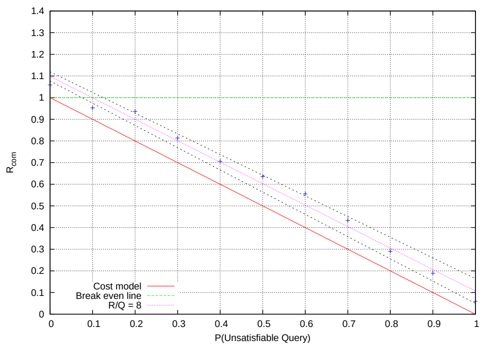
(b) Restrictions per Query $R/Q = 2$



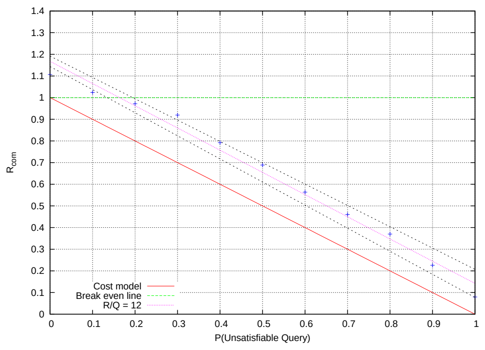
(c) Restrictions per Query $R/Q = 3$



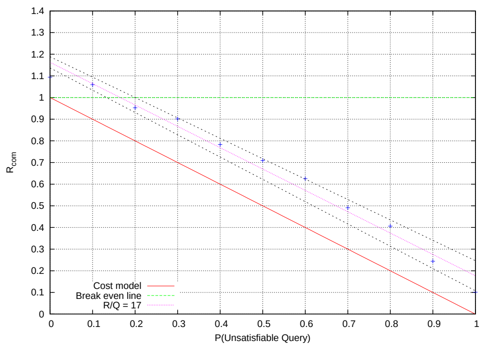
(d) Restrictions per Query $R/Q = 5$



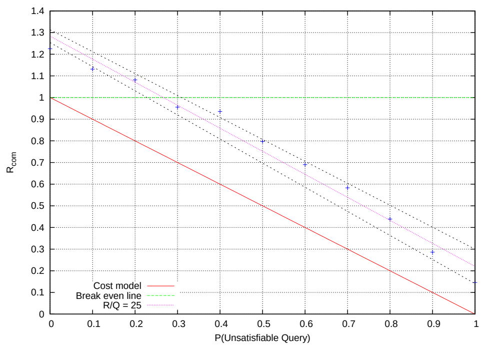
(e) Restrictions per Query $R/Q = 8$



(f) Restrictions per Query $R/Q = 12$

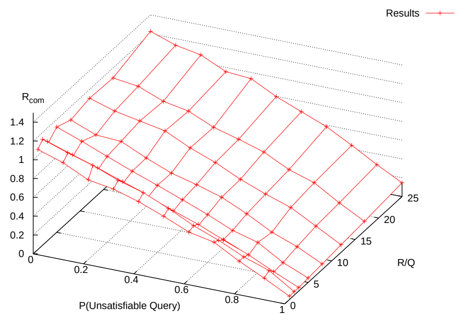


(g) Restrictions per Query $R/Q = 17$

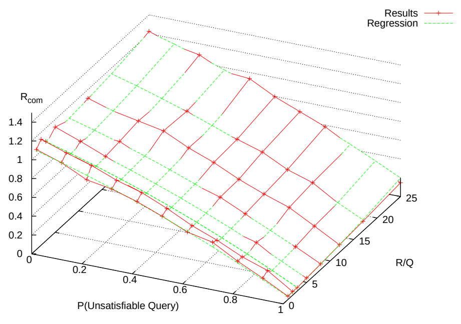


(h) Restrictions per Query $R/Q = 25$

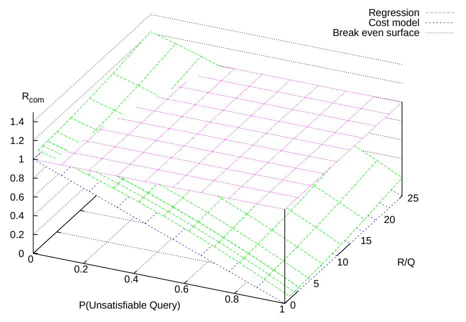
Figure A.12: **Ratio R_{com} vs Probability of Unsatisfiable Query P (indexed)**: Figures A.12(a) to A.12(h) show the increasing penalty paid by the semantic optimizer as query complexity increases. As the number of restrictions per query (R/Q) increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



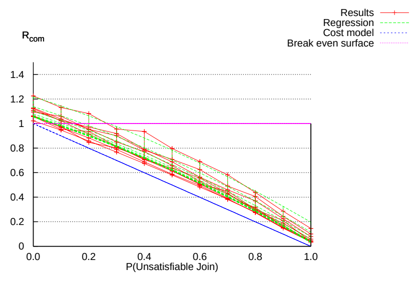
(a) R_{com} surface for $R/Q = 1$ to 25.



(b) R_{com} surface with regression surface.

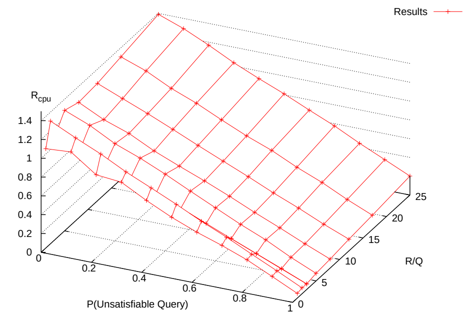


(c) Regression, “cost model” and “break even”surfaces.

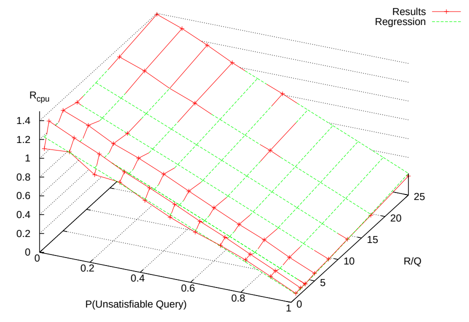


(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

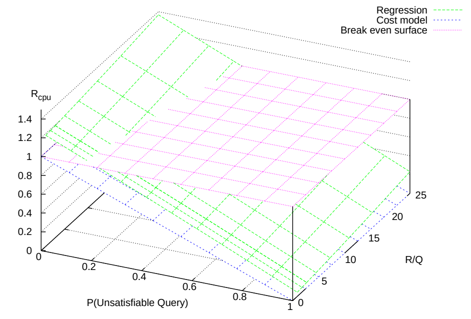
Figure A.13: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.12 as a R_{com} surface. As the number of Restrictions per Query R/Q increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $P = 10\%$, positive optimization is achieved when there is up to five restrictions per query; i.e., $R/Q \leq 5$. Number of table rows $Rows = 1,000,000$.



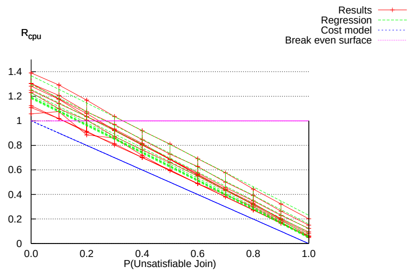
(a) R_{cpu} surface for R/Q = 1 to 25



(b) R_{cpu} surface with regression surface.

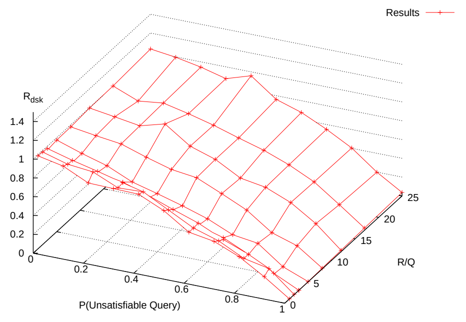


(c) Regression, “cost model” and “break even”surfaces.

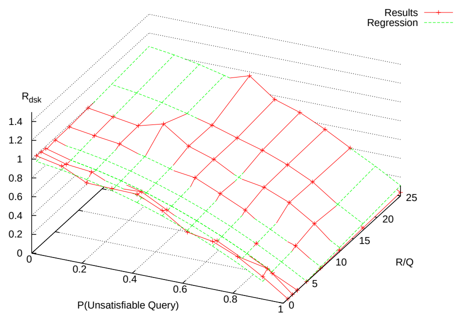


(d) R_{cpu}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

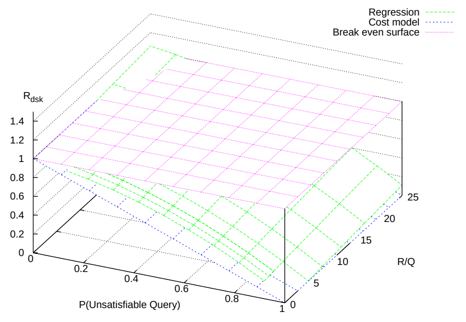
Figure A.14: **Ratio R_{cpu} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.12 as a R_{cpu} surface. As the number of restrictions per query (R/Q) increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



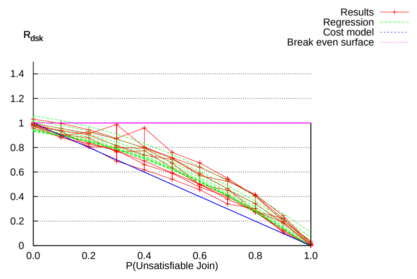
(a) R_{dsk} surface for R/Q = 1 to 25



(b) R_{dsk} surface with regression surface.

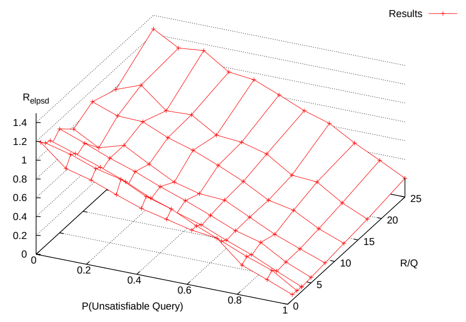


(c) Regression, "cost model" and "break even"surfaces.

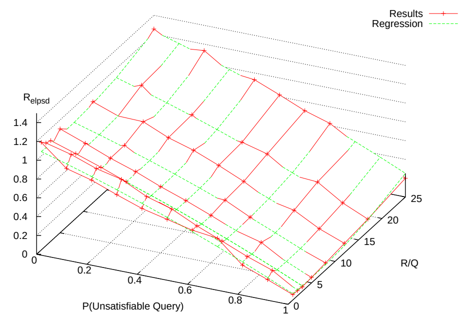


(d) R_{dsk}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

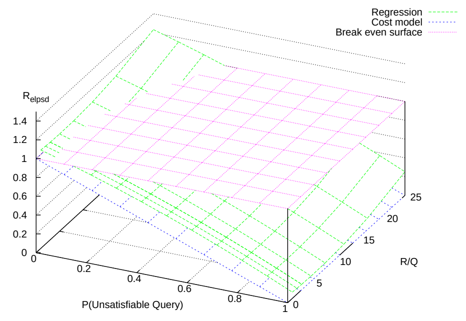
Figure A.15: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.12 as a R_{dsk} surface. As the number of restrictions per query (R/Q) increases from 1 to 25, positive optimization is maintained up to an $R/Q \approx 20$. Number of table rows $Rows = 1,000,000$.



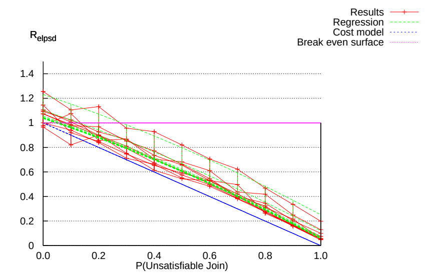
(a) R_{elpsd} surface for $R/Q = 1$ to 25



(b) R_{elpsd} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{elpsd}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure A.16: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.12 as a R_{elpsd} surface. As the number of restrictions per query (R/Q) increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.

A.5 Indexed Unsatisfiable Queries – Varying Intervals per Restriction

This section contains a full analysis of the results reported in Section 6.6. The objective of these experiments is to establish the dependence of the *gain in query efficiency* on the *probability of an unsatisfiable query* and the *number of intervals per restriction.* In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable query P
- Number of intervals per restriction I/R. Each query has a single restriction.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in query restrictions are indexed with a "normal" B-tree index.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and I/R
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (intervals per restriction I/R) disappears

A.5.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.17 plots R_{com} vs Probability of Unsatisfiable Query P and shows the penalty paid by the semantic optimizer, as the number of intervals comprising the single restriction increases, is balanced by the increased processing time required by the normal SQL optimizer. Therefore the ratio R_{com} rises only slowly as I/R increases from 1 to 25.

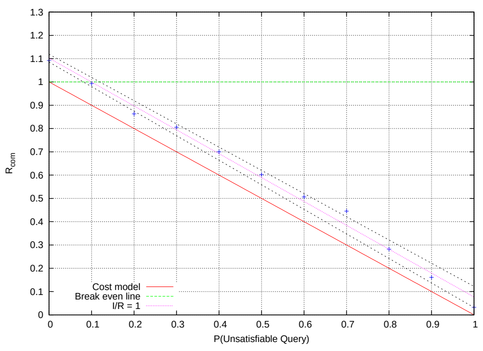
Figure A.18 plots R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R and summarises the results as a R_{com} surface. For low I/R, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the "cost model surface”. As I/R rises, while the pre-processing cost becomes significant, this is balanced by the increased processing time required by the normal SQL optimizer. The net result is that the combined ratio R_{com} hardly varies with increasing I/R.

A.5.2 Individual Cost Metric Ratios

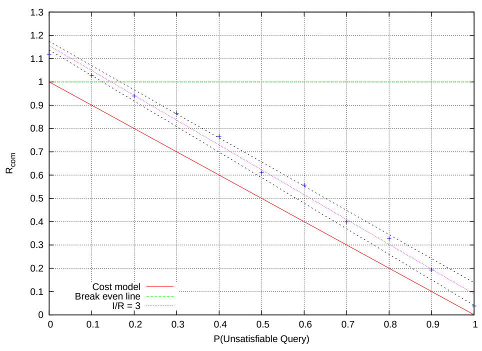
The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.5.

Figure	**Results Presented**
A.19	R_{cpu} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R
A.20	R_{dsk} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R
A.21	R_{elpsd} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R

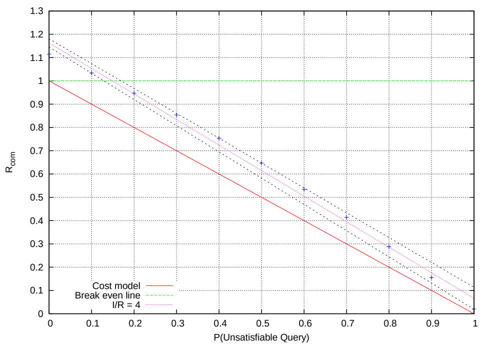
Table A.5: **Location of summary results** for the three individual cost metric ratios displaying R_{cost} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R.



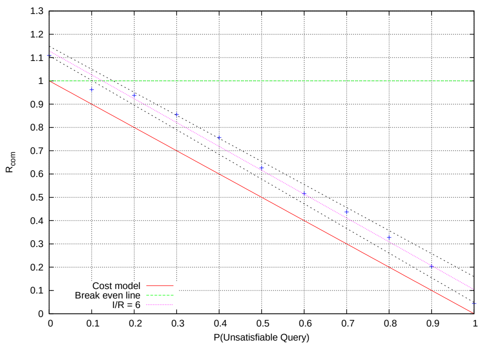
(a) Intervals per Restriction $I/R = 1$



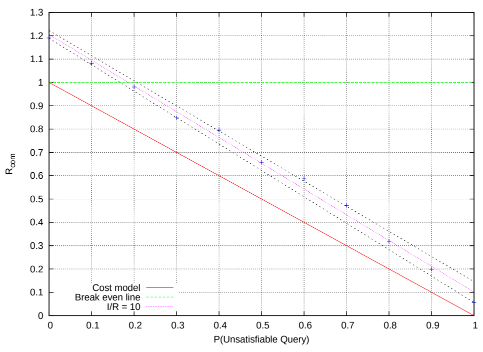
(b) Intervals per Restriction $I/R = 3$



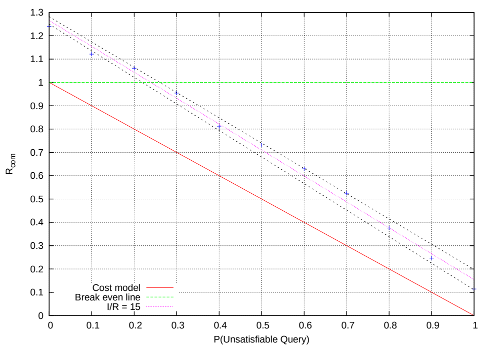
(c) Intervals per Restriction $I/R = 4$



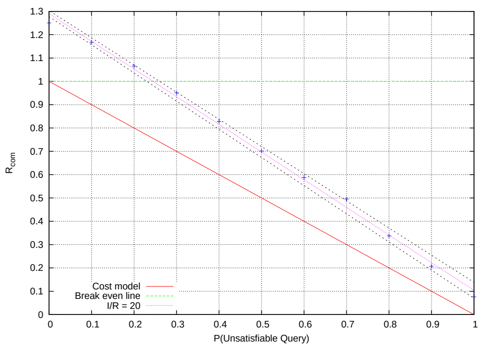
(d) Intervals per Restriction $I/R = 6$



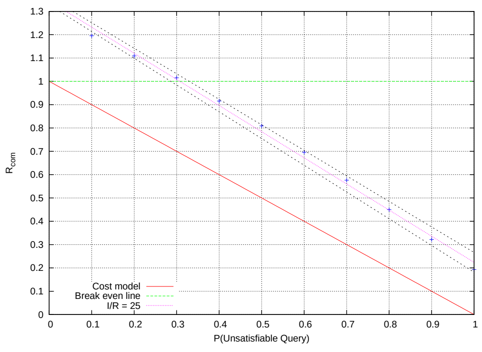
(e) Intervals per Restriction $I/R = 10$



(f) Intervals per Restriction $I/R = 15$

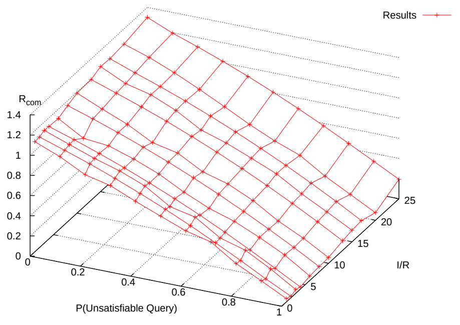


(g) Intervals per Restriction $I/R = 20$

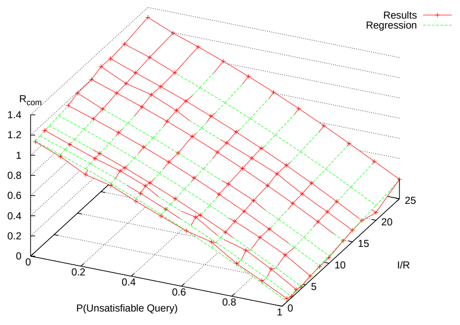


(h) Intervals per Restriction $I/R = 25$

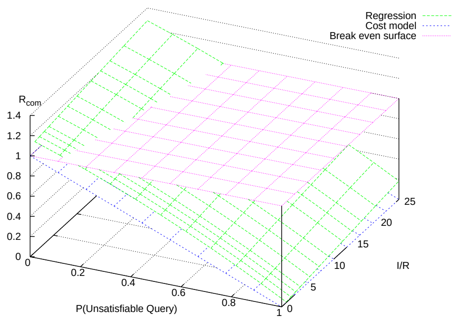
Figure A.17: **Ratio R_{com} vs Probability of Unsatisfiable Query P (indexed)**: As the number of Intervals per Restriction I/R increases from 1 to 25, ratio R_{com} increases slowly. For $P > 0.15$, positive optimization is achieved throughout the whole range. Number of table rows $Rows = 1,000,000$.



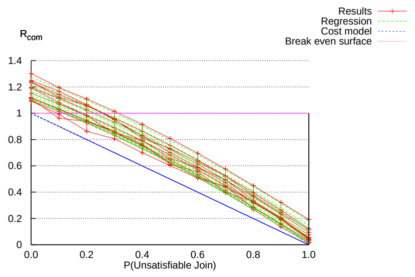
(a) R_{com} surface for $I/R = 1$ to 25



(b) R_{com} surface with regression surface.

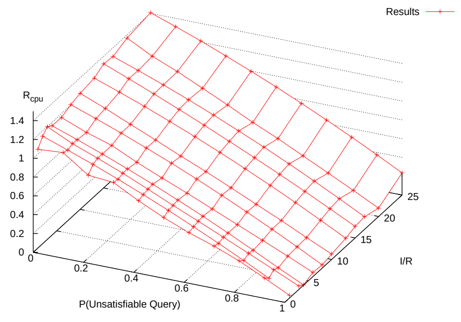


(c) Regression, “cost model” and “break even”surfaces.

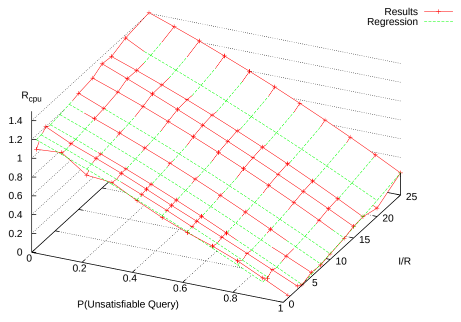


(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

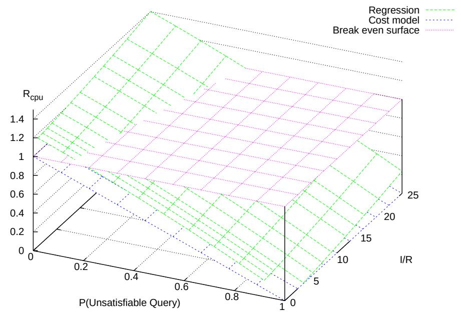
Figure A.18: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.17 as a R_{com} surface. As the number of Intervals per Restriction I/R increases from 1 to 25, ratio R_{com} hardly increases. For $P = 5\%$, positive optimization is achieved throughout the whole range. Number of table rows $Rows = 1,000,000$.



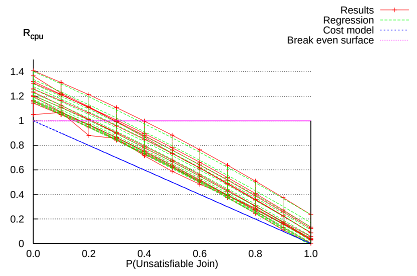
(a) R_{cpu} surface for I/R = 1 to 25



(b) R_{cpu} surface with regression surface.

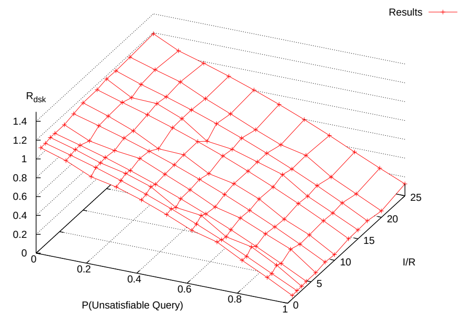


(c) Regression, “cost model” and “break even”surfaces.

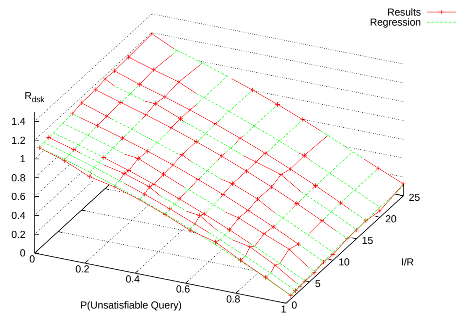


(d) R_{cpu}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

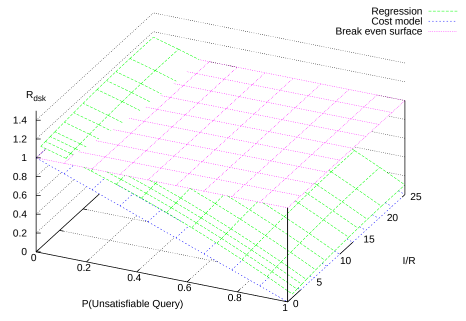
Figure A.19: **Ratio R_{cpu} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.17 as a R_{cpu} surface. As the number of restrictions per query (I/R) increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



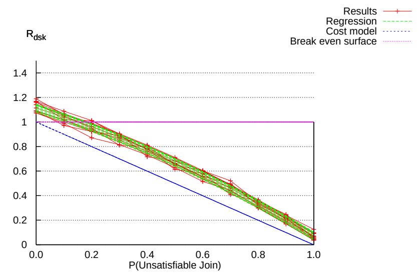
(a) R_{dsk} surface for I/R = 1 to 25



(b) R_{dsk} surface with regression surface.

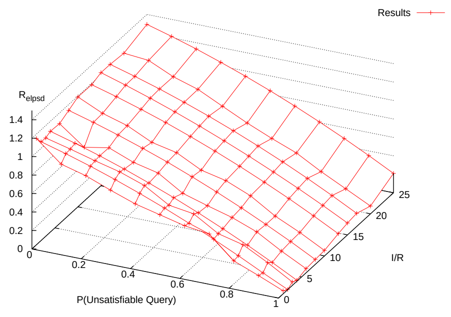


(c) Regression, "cost model" and "break even"surfaces.

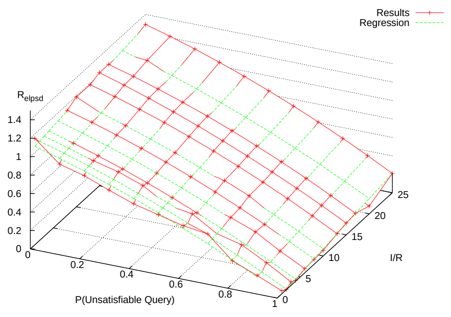


(d) R_{dsk}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

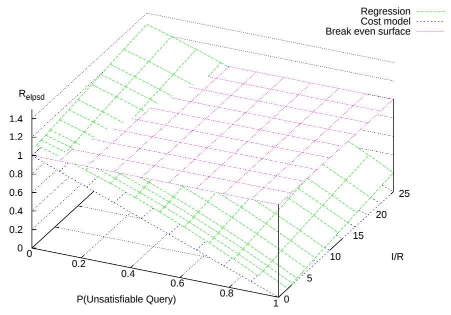
Figure A.20: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.17 as a R_{dsk} surface. As the number of restrictions per query (I/R) increases from 1 to 25, positive optimization is maintained up to an $I/R \approx 25$. Results for disk i/o typically exhibit more variation than the other metric ratios. Number of table rows $Rows = 1,000,000$.



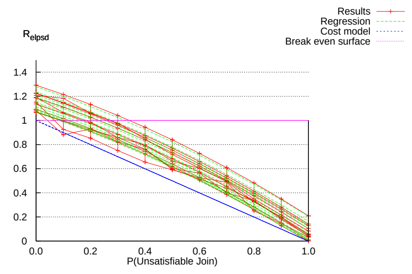
(a) R_{elpsd} surface for $I/R = 1$ to 25



(b) R_{elpsd} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{elpsd}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure A.21: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.17 as a R_{elpsd} surface. As the number of restrictions per query (I/R) increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. Results for elapsed time typically exhibit more variation than the other metric ratios. Number of table rows $Rows = 1,000,000$.

A.6 Indexed Unsatisfiable Joins

This section contains a full analysis of the results reported in Section 6.4. The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and *relative table size*. The methodology is identical to the experiments reported above in Appendix A.3, except that we submit batches of equi-joins between two tables rather than simple queries against a single table.

In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P
- Relative table size, denoted by the number of table rows $Rows$.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. Each join consists of a single join clause citing the equi-join columns plus a single restriction defined by one interval.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and $Rows$
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with “cost model” and “break even” surfaces
- The cost metric ratio surface R_{cost}, regression surface, “cost model” and “break even” surfaces viewed by looking directly into the XZ plane such that the Y axis (relative table size $Rows$) disappears

A.6.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.22 plots R_{com} vs Probability of Unsatisfiable Join P and shows how the R_{com} ratio stays relatively consistent as table size increases from $Rows = 1,000$ to $1,000,000$. The results conform quite closely to the cost model, but not as closely as for the equivalent experiments with simple queries (see Figure A.7).

Figure A.23(a) plots R_{com} vs Probability of Unsatisfiable Join P vs Relative Table Size $Rows$ and summarises the results as a R_{com} surface. Figure A.23(b) shows the same surface along with the regression surface. Figure A.23(c) compares the idealised regression, "cost model" and the "break even" surfaces. Figure A.23(d) is exactly the same projection, but viewed by looking directly into the XZ plane. This clearly shows the R_{com} surface sits just above the "cost model surface", indicating results deviate very little from the predicted, with positive optimization occurring when $P > 20\%$, across four orders of magnitude of table size.

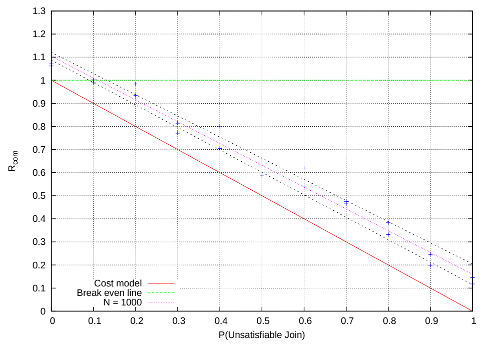
A.6.2 Individual Cost Metric Ratios

The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.6.

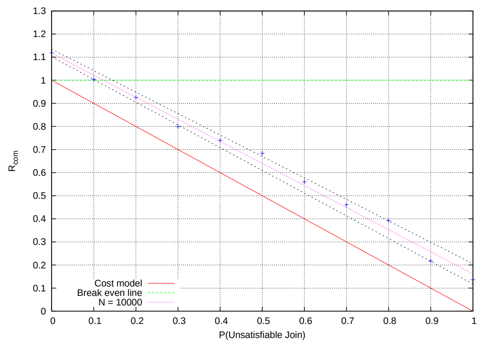
Figure	**Results Presented**
A.24	R_{cpu} vs Probability of Unsatisfiable Join P vs Rows N
A.25	R_{dsk} vs Probability of Unsatisfiable Join P vs Rows N
A.26	R_{elpsd} vs Probability of Unsatisfiable Join P vs Rows N

Table A.6: **Location of summary results** for the three individual cost metric ratios displaying R_{cnt} vs Probability of Unsatisfiable Join P vs Rows N .

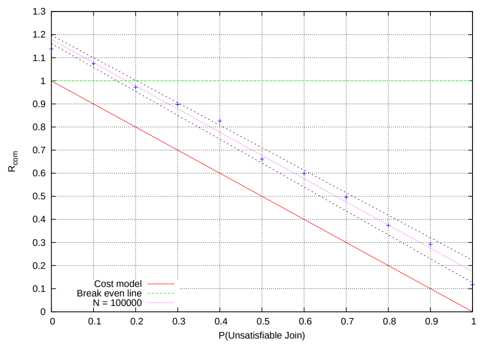
The individual cost metric ratios show more variation than the combined cost metric ratio R_{com} results displayed in Section A.6.1. This is described in detail in Chapter 5. We note however that the results show a consistent efficiency gain for the semantically optimized queries and a close correspondence to the results predicted from our cost model.



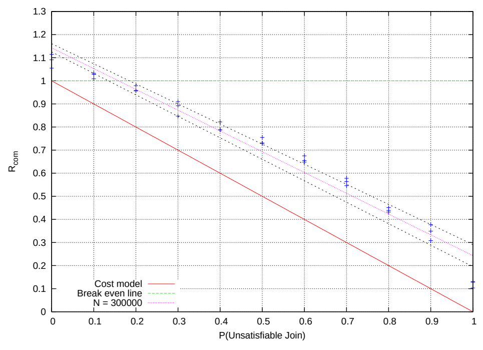
(a) Table rows `N = 1,000`



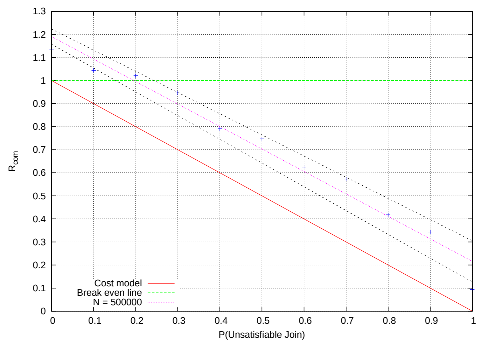
(b) Table rows `N = 10,000`



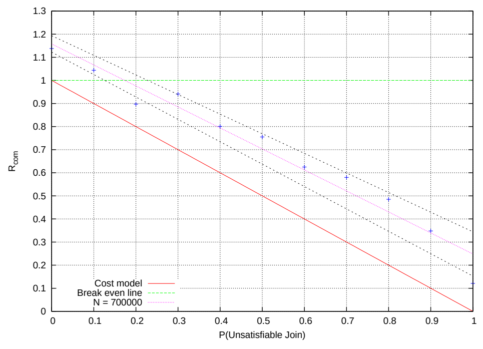
(c) Table rows `N = 100,000`



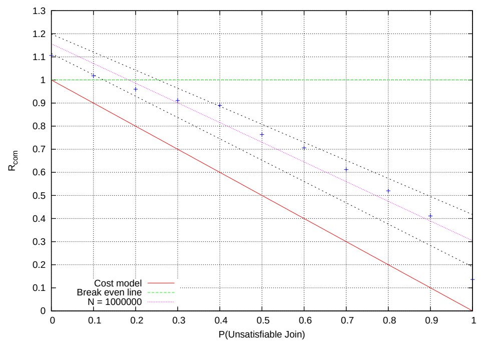
(d) Table rows `N = 300,000`



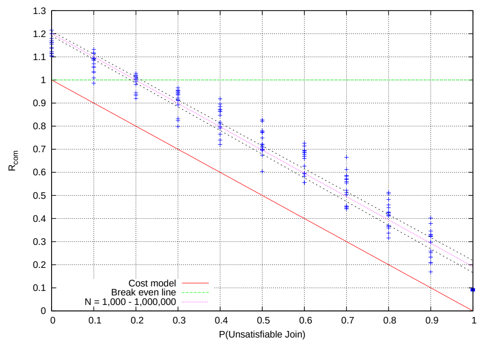
(e) Table rows `N = 500,000`



(f) Table rows `N = 700,000`

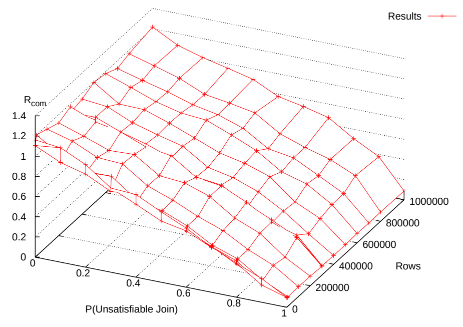


(g) Table rows `N = 1,000,000`

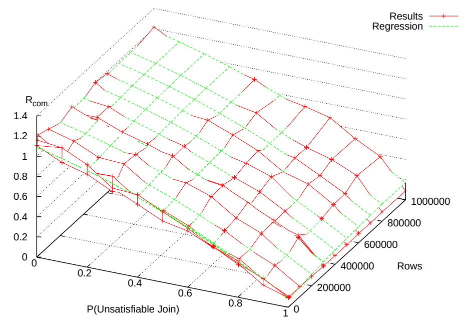


(h) Table rows `N = 100` to `1,000,000`

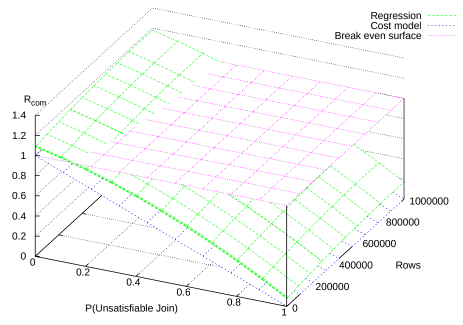
Figure A.22: **Ratio R_{com} vs Probability of Unsatisfiable Join P (indexed)**: Figures A.22(a) to A.22(h) show the R_{com} ratio stays relatively consistent as table size increases from $Rows = 1,000$ to $1,000,000$. The results conform quite closely to the cost model, but not as closely as for the equivalent experiments with simple queries (see Figure A.7). Figure A.22(h) combines all results into a single graph and this highlights the greater spread of results than for the equivalent experiments with simple queries. The *combined* ratio R_{com} is the average of the other three cost metric ratios which we interpret as the overall join cost.



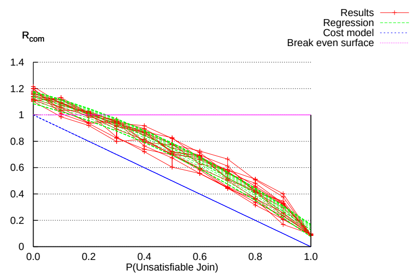
(a) R_{com} surface for `100` to `1,000,000` rows



(b) R_{com} surface with regression surface.

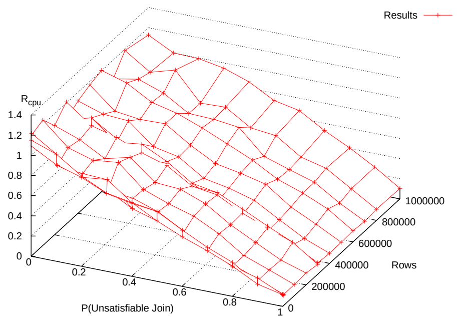


(c) Regression, “cost model” and “break even”surfaces.

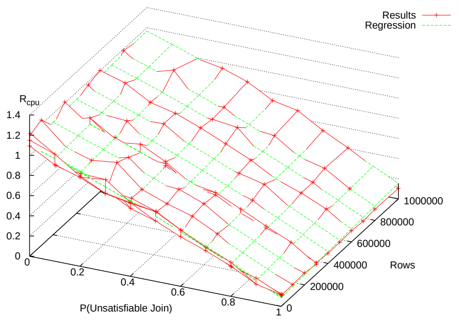


(d) R_{com}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

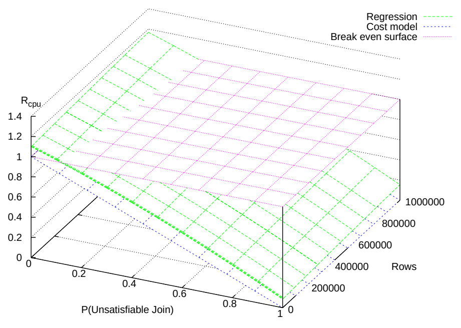
Figure A.23: **Ratio** R_{com} **vs Probability of Unsatisfiable Join** P **vs Relative Table Size** *Rows* **(indexed)**: These figures summarise the results presented above in Figure A.22 as a R_{com} surface. Figures A.23(b) and A.23(c) show R_{com} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.2$.



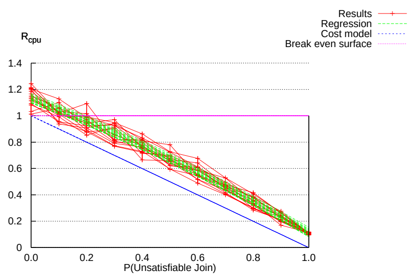
(a) R_{cpu} surface for `100` to `1,000,000` rows



(b) R_{cpu} surface with regression surface.

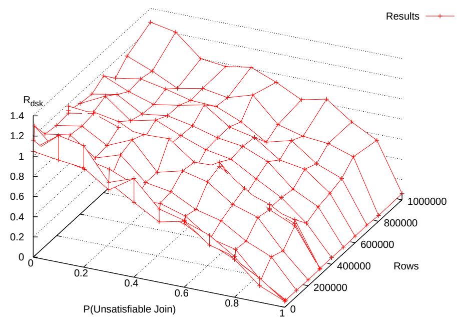


(c) Regression, “cost model” and “break even”surfaces.

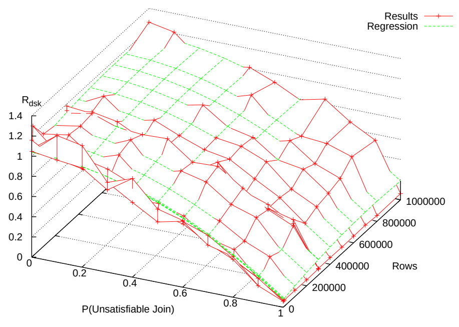


(d) R_{cpu}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

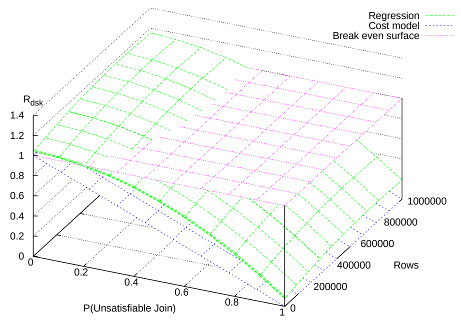
Figure A.24: **Ratio R_{cpu} vs Probability of Unsatisfiable Join P vs Relative Table Size *Rows* (indexed)**: These figures summarise the results presented above in Figure A.22 as a R_{cpu} surface. Figures A.24(b) and A.24(c) show R_{cpu} surface sits just above the “cost model surface”, indicating results deviate very little from the predicted. We have positive optimization across four orders of magnitude of table size when $P > 0.15$.



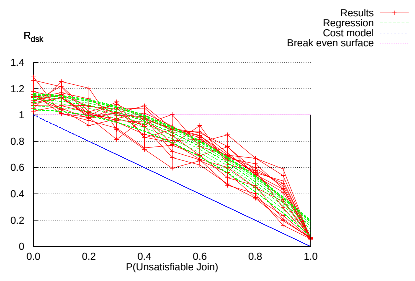
(a) R_{dsk} surface for `100` to `1,000,000` rows



(b) R_{dsk} surface with regression surface.

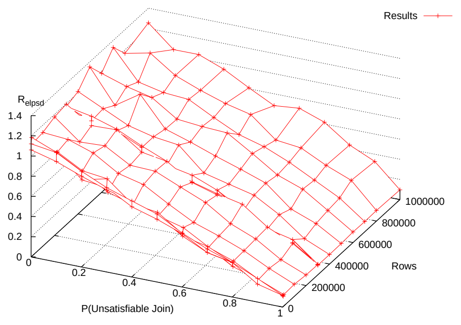


(c) Regression, “cost model” and “break even”surfaces.

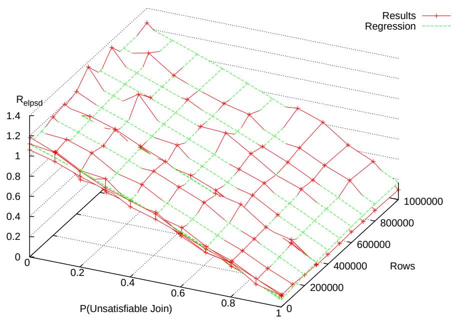


(d) R_{dsk}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

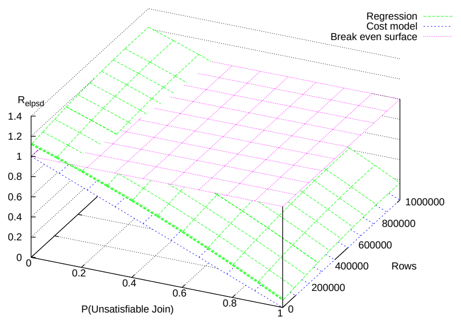
Figure A.25: **Ratio R_{dsk} vs Probability of Unsatisfiable Join P vs Relative Table Size *Rows* (indexed)**: The R_{dsk} surface sits just above the “cost model surface” and is clearly influenced by relative table size. With regrad to disk i/o, we require $P > 0.2$ in order to break even. In comparison with the equivalent results for simple queries (see Figure A.10), optimization is significantly degraded by disk i/o for joins.



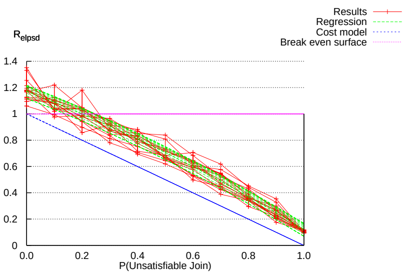
(a) R_{elpsd} surface for `100` to `1,000,000` rows



(b) R_{elpsd} surface with regression surface.



(c) Regression, “cost model” and “break even”surfaces.



(d) R_{elpsd}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

Figure A.26: **Ratio R_{elpsd} vs Probability of Unsatisfiable Join P vs Relative Table Size *Rows* (indexed)**: These figures summarise the results presented above in Figure A.22 as a R_{elpsd} surface. Figures A.26(b) and A.26(c) show R_{elpsd} surface sits just above the “cost model surface” and is little influenced by increasing table size. We have positive optimization across four orders of magnitude of table size when $P > 0.2$.

A.7 Indexed Unsatisfiable Joins – Varying Restrictions per Join

This section contains a full analysis of the results reported in Section 6.5. The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and the *number of restrictions per join*. In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P
- Number of restrictions per join R/Q. Each restriction is defined by a single interval.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in join restrictions are indexed with a "normal" B-tree index.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and R/Q
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (restrictions per query R/Q) disappears

A.7.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.27 plots R_{com} vs Probability of Unsatisfiable Query P and shows how this ratio varies with increasing numbers of restrictions per join (R/Q). Each restriction is defined by a single interval. The results show R_{com} increases as R/Q increases. This is what we expect since the cost of semantically pre-processing the join rises with increasing join complexity.

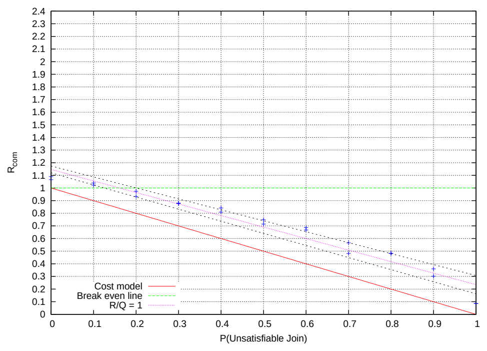
Figure A.28 plots R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q and summarises the results as a R_{com} surface. For low R/Q, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the "cost model surface". However as R/Q rises, the pre-processing cost becomes significant and we require a greater proportion of unsatisfiable queries to make semantic optimization worthwhile.

A.7.2 Individual Cost Metric Ratios

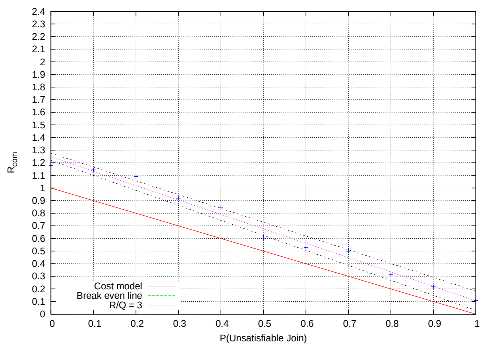
The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.7.

Figure	**Results Presented**
A.29	R_{cpu} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q
A.30	R_{dsk} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q
A.31	R_{elpsd} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q

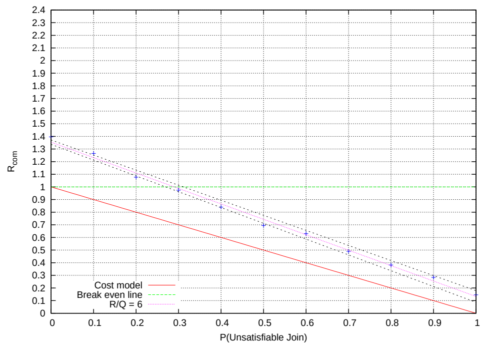
Table A.7: **Location of summary results** for the three individual cost metric ratios displaying R_{cost} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q.



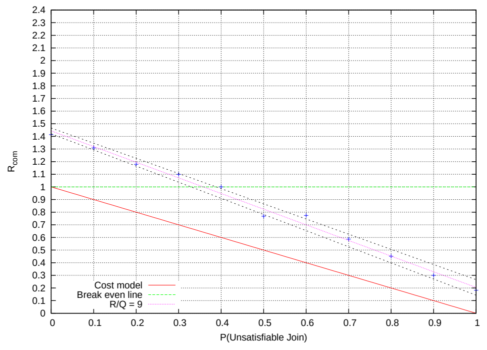
(a) Restrictions per Query $R/Q = 1$



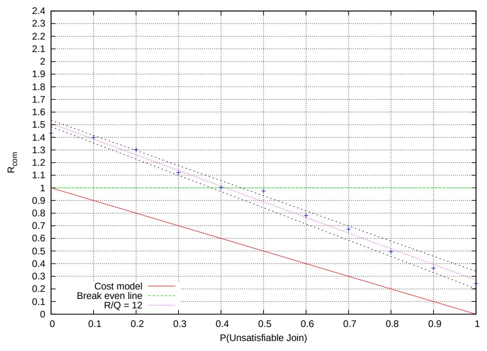
(b) Restrictions per Query $R/Q = 3$



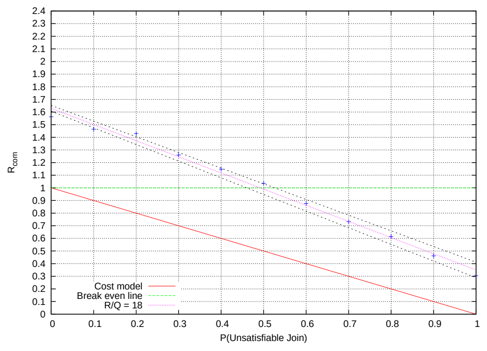
(c) Restrictions per Query $R/Q = 6$



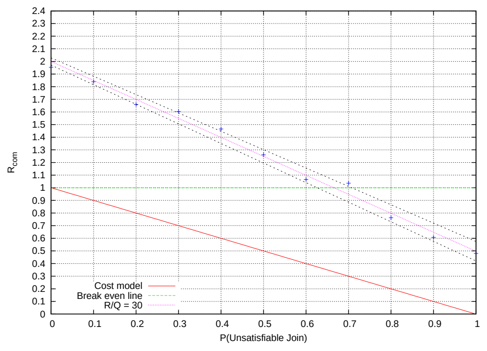
(d) Restrictions per Query $R/Q = 9$



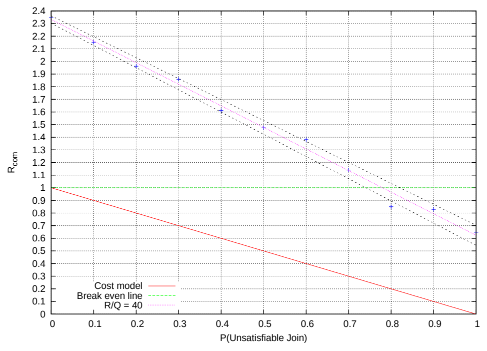
(e) Restrictions per Query $R/Q = 12$



(f) Restrictions per Query $R/Q = 18$

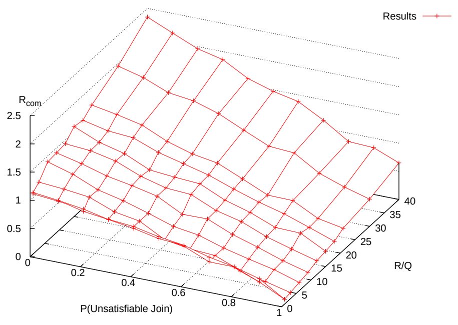


(g) Restrictions per Query $R/Q = 30$

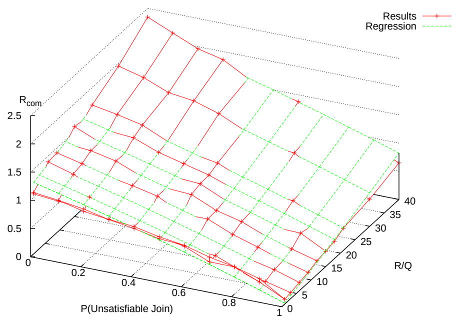


(h) Restrictions per Query $R/Q = 40$

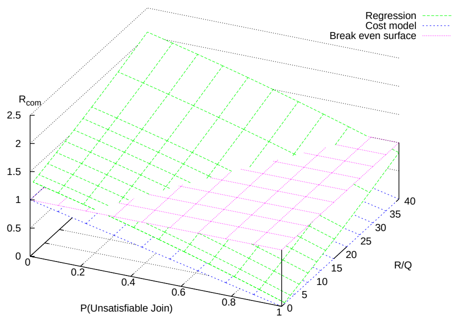
Figure A.27: **Ratio R_{com} vs Probability of Unsatisfiable Query P (indexed)**: Figures A.27(a) to A.27(h) show the increasing penalty paid by the semantic optimizer as join complexity increases. As the number of restrictions per join (R/Q) increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



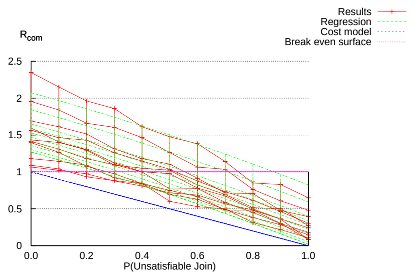
(a) R_{com} surface for $R/Q = 1$ to 40.



(b) R_{com} surface with regression surface.

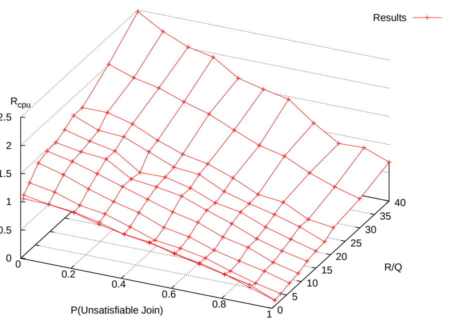


(c) Regression, "cost model" and "break even"surfaces.

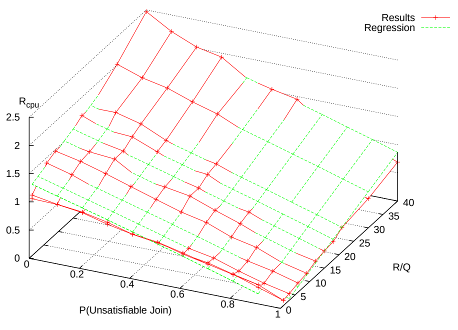


(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

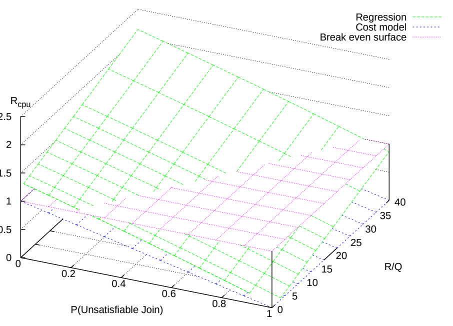
Figure A.28: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.27 as a R_{com} surface. As the number of Restrictions per Query R/Q increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. For $P = 0.2$, positive optimization is achieved when there is up to five restrictions per join; i.e., $R/Q \leq 5$. Number of table rows $Rows = 1,000,000$.



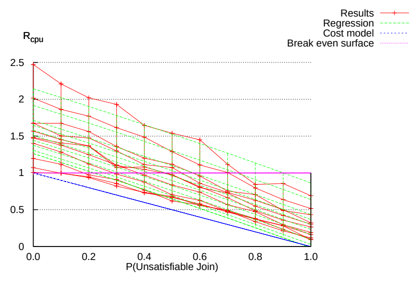
(a) R_{cpu} surface for $R/Q = 1$ to 40



(b) R_{cpu} surface with regression surface.

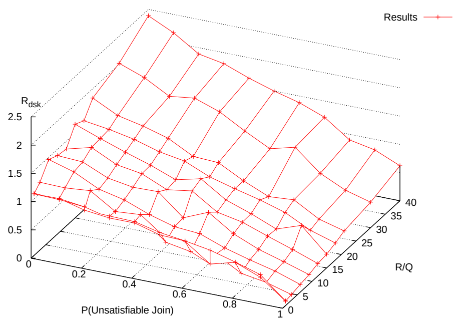


(c) Regression, "cost model" and "break even"surfaces.

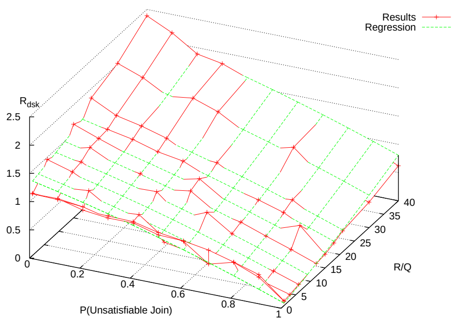


(d) R_{cpu}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

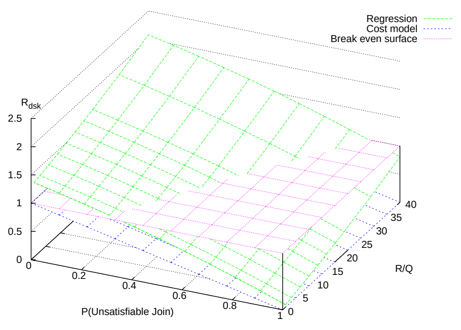
Figure A.29: **Ratio R_{cpu} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.27 as a R_{cpu} surface. As the number of restrictions per join (R/Q) increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



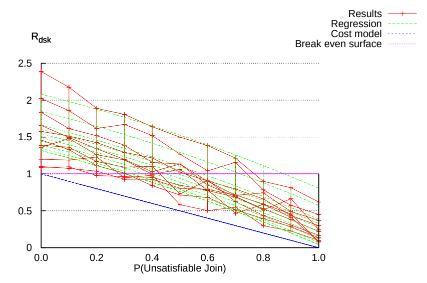
(a) R_{dsk} surface for $R/Q = 1$ to 40



(b) R_{dsk} surface with regression surface.

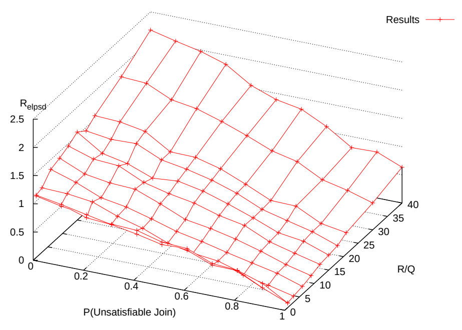


(c) Regression, “cost model” and “break even”surfaces.

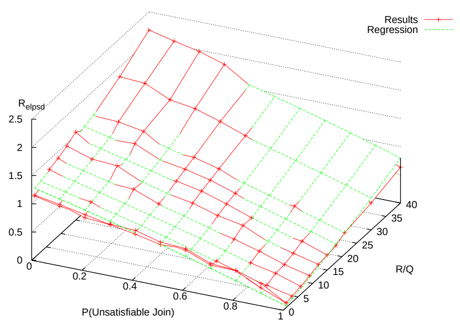


(d) R_{dsk}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

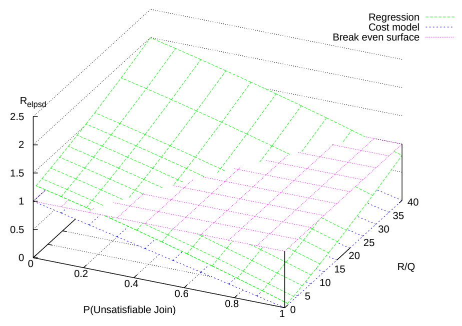
Figure A.30: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Restrictions per Query R/Q (indexed)**: These figures summarise the results presented above in Figure A.27 as a R_{dsk} surface. As the number of restrictions per join (R/Q) increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$. Number of table rows $Rows = 1,000,000$.



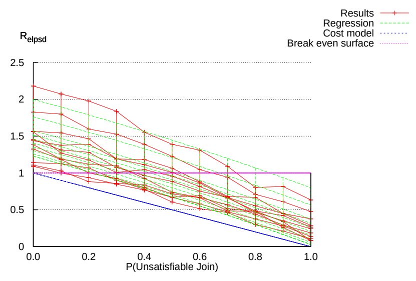
(a) R_{elpsd} surface for $R/Q = 1$ to 40



(b) R_{elpsd} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{elpsd}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure A.31: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Restrictions per Query** R/Q (indexed): These figures summarise the results presented above in Figure A.27 as a R_{elpsd} surface. As the number of restrictions per join (R/Q) increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.

A.8 Indexed Unsatisfiable Joins – Varying Intervals per Restriction

This section contains a full analysis of the results reported in Section 6.6. The objective of these experiments is to establish the dependence of the *gain in join efficiency* on the *probability of an unsatisfiable join* and the *number of intervals per restriction.* In this series of experiments, we have two independent variables:

- Probability of an unsatisfiable join P
- Number of intervals per restriction I/R. Each join has a single restriction.

The dependent variable is the cost ratio R_{cost} where R_{cost} is one of the ratios defined in Table A.1. All results are for tables with number of rows $Rows = 1,000,000$. All columns cited in join restrictions are indexed with a "normal" B-tree index.

We begin with a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results. This is followed by summary results for the three individual cost metric ratios. In the case of summary graphs, we present the same four variations depicting:

- The cost metric ratio surface R_{cost} plotted against the two independent variables P and I/R
- The cost metric ratio surface R_{cost} with the regression surface superimposed
- The regression surface with "cost model" and "break even" surfaces
- The cost metric ratio surface R_{cost}, regression surface, "cost model" and "break even" surfaces viewed by looking directly into the XZ plane such that the Y axis (intervals per restriction I/R) disappears

A.8.1 Combined Ratio: R_{com}

The following is a detailed analysis of the combined metric ratio R_{com}, presenting both individual result graphs and summary results.

Figure A.32 plots R_{com} vs Probability of Unsatisfiable Query P and shows the penalty paid by the semantic optimizer, as the number of intervals comprising the single restriction increases, is balanced by the increased processing time required by the normal SQL optimizer. Therefore the ratio R_{com} rises only modestly as I/R increases from 1 to 25.

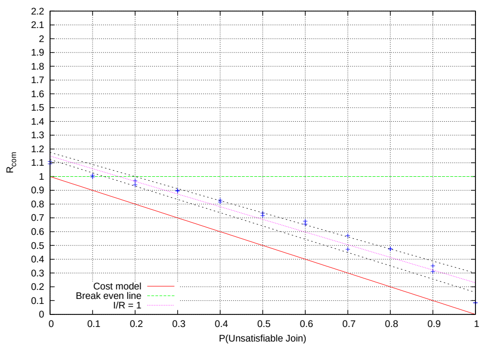
Figure A.33 plots R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R and summarises the results as a R_{com} surface. For low I/R, semantic pre-processing incurs little overhead and the R_{com} surface sits just above the "cost model surface". As I/R rises, while the pre-processing cost becomes significant, this is balanced by the increased processing time required by the normal SQL optimizer. The net result is that the combined ratio R_{com} hardly varies with increasing I/R.

A.8.2 Individual Cost Metric Ratios

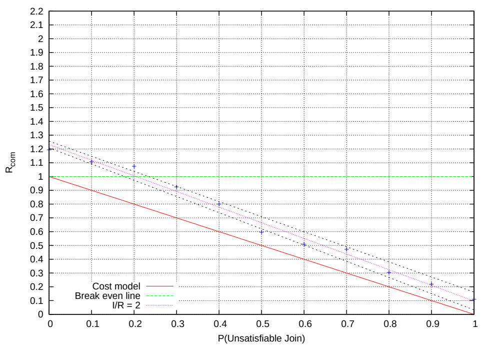
The remaining graphs in this section show summary results for the three individual cost metric ratios, located as set out in Table A.8.

Figure	**Results Presented**
A.34	R_{cpu} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R
A.35	R_{dsk} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R
A.36	R_{elpsd} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R

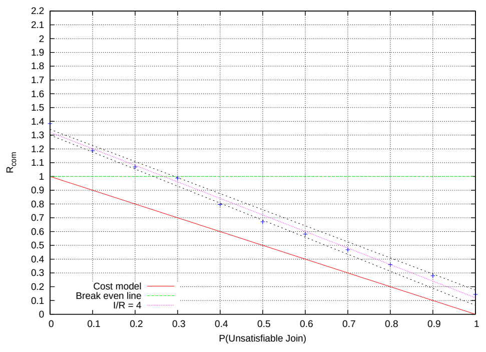
Table A.8: **Location of summary results** for the three individual cost metric ratios displaying R_{cost} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R.



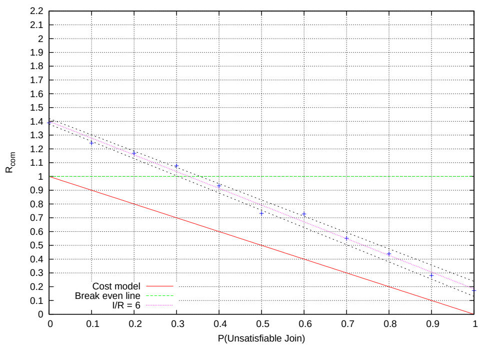
(a) Intervals per Restriction $I/R = 1$



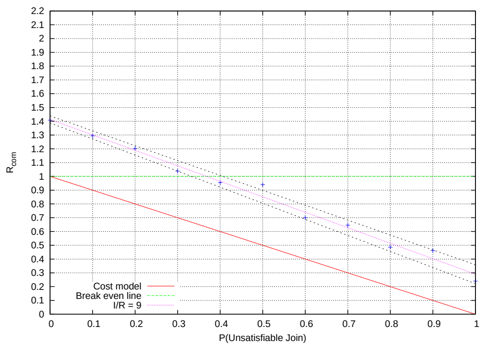
(b) Intervals per Restriction $I/R = 2$



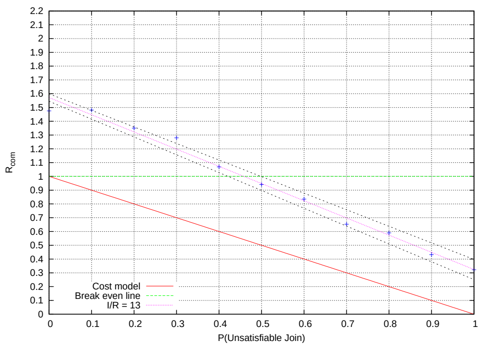
(c) Intervals per Restriction $I/R = 4$



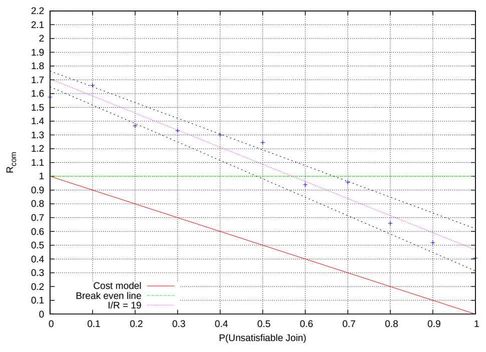
(d) Intervals per Restriction $I/R = 6$



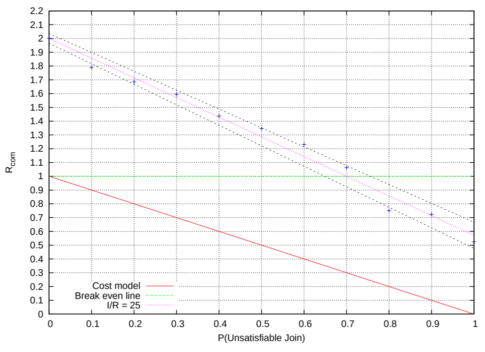
(e) Intervals per Restriction $I/R = 9$



(f) Intervals per Restriction $I/R = 13$

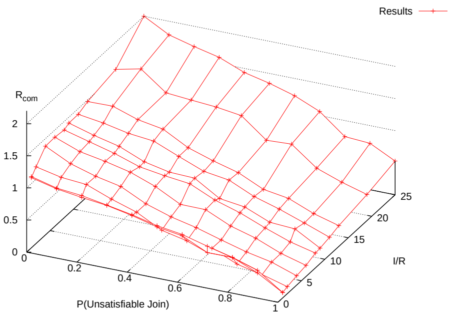


(g) Intervals per Restriction $I/R = 19$

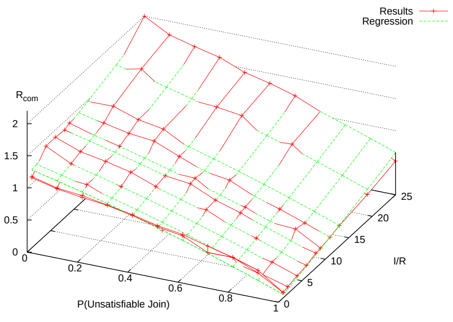


(h) Intervals per Restriction $I/R = 25$

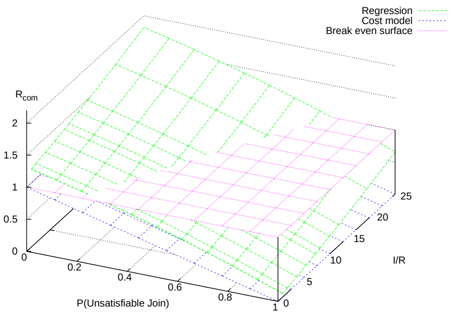
Figure A.32: **Ratio R_{com} vs Probability of Unsatisfiable Query P (indexed)**: Figures A.32(a) to A.32(h) show the increasing penalty paid by the semantic optimizer as join complexity increases. As the number of intervals per restriction (I/R) increases from 1 to 40, a greater proportion of unsatisfiable queries is required in order to break even. Number of table rows $Rows = 1,000,000$.



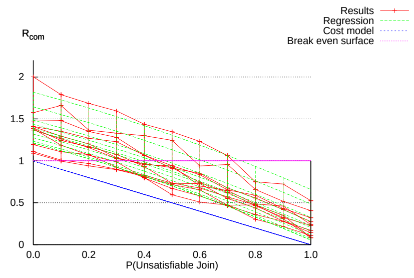
(a) R_{com} surface for $I/R = 1$ to 25



(b) R_{com} surface with regression surface.

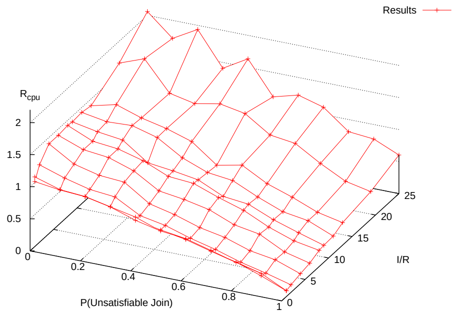


(c) Regression, "cost model" and "break even"surfaces.

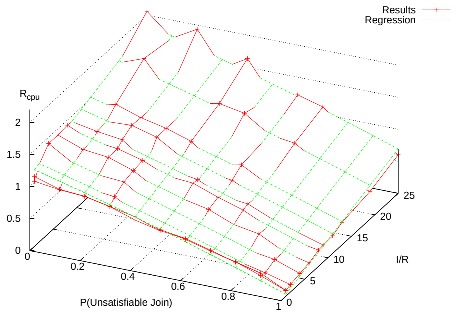


(d) R_{com}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

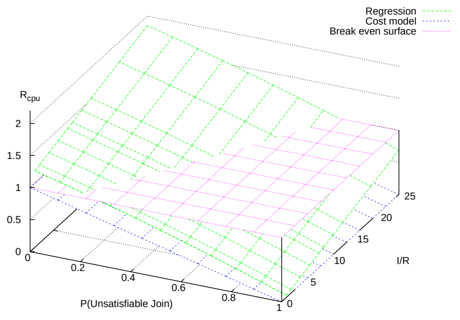
Figure A.33: **Ratio R_{com} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.32 as a R_{com} surface. As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.3$ to achieve positive optimization. Number of table rows $Rows = 1,000,000$.



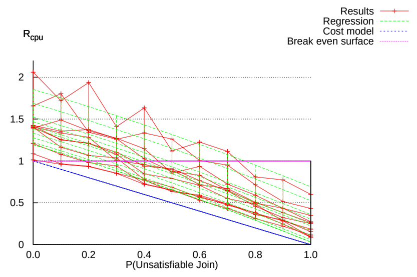
(a) R_{cpu} surface for $I/R = 1$ to 25



(b) R_{cpu} surface with regression surface.

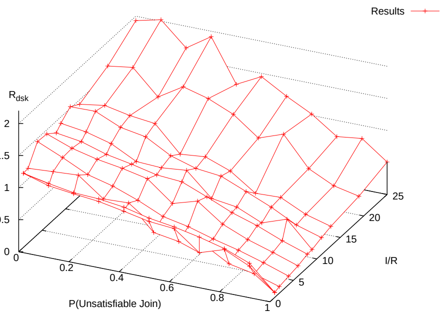


(c) Regression, “cost model” and “break even”surfaces.

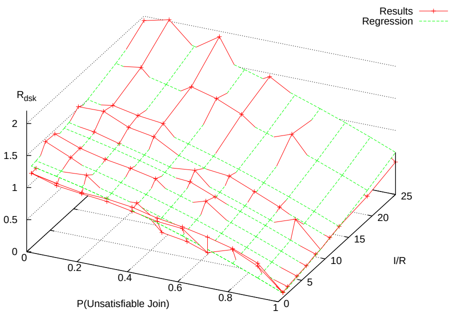


(d) R_{cpu}, regression, “cost model” and “break even” surfaces, looking directly into the XZ plane.

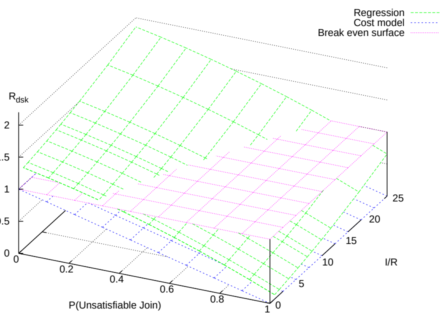
Figure A.34: **Ratio R_{cpu} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.32 as a R_{cpu} surface. As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.25$ to achieve positive optimization. Number of table rows $Rows = 1,000,000$.



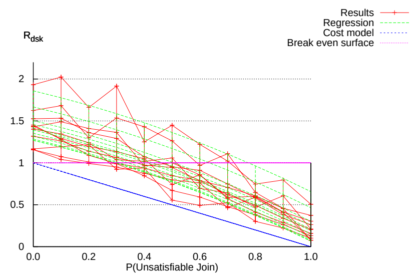
(a) R_{dsk} surface for $I/R = 1$ to 25



(b) R_{dsk} surface with regression surface.

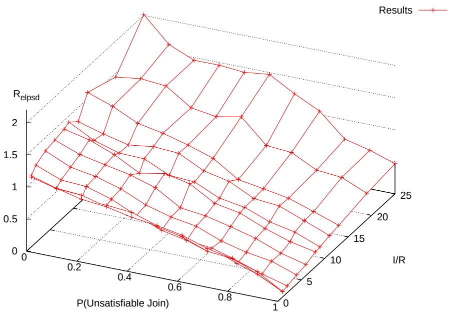


(c) Regression, "cost model" and "break even"surfaces.

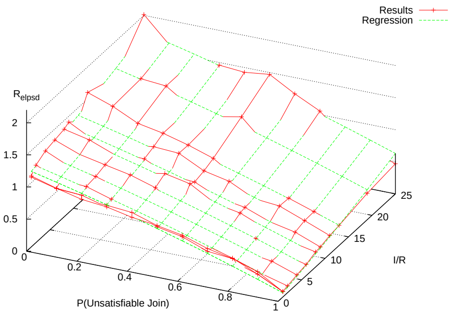


(d) R_{dsk}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

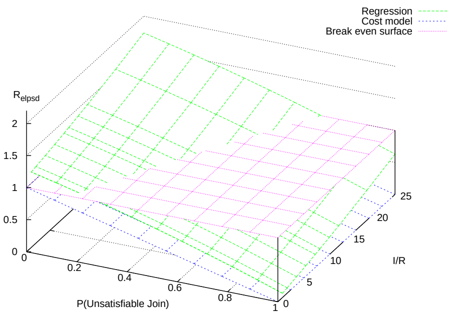
Figure A.35: **Ratio R_{dsk} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.32 as a R_{dsk} surface. As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.3$ to achieve positive optimization. Number of table rows $Rows = 1,000,000$.



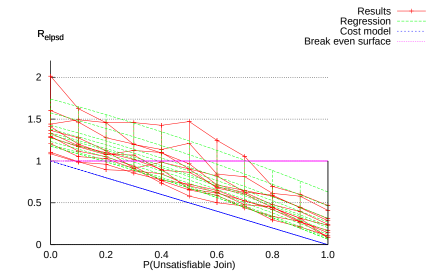
(a) R_{elpsd} surface for $I/R = 1$ to 25



(b) R_{elpsd} surface with regression surface.



(c) Regression, "cost model" and "break even"surfaces.



(d) R_{elpsd}, regression, "cost model" and "break even" surfaces, looking directly into the XZ plane.

Figure A.36: **Ratio R_{elpsd} vs Probability of Unsatisfiable Query P vs Intervals per Restriction I/R (indexed)**: These figures summarise the results presented above in Figure A.32 as a R_{elpsd} surface. As the number of intervals per restriction I/R increases from 1 to 25, a greater proportion of unsatisfiable queries is required in order to break even. For $I/R \leq 5$, we require just on $P = 0.2$ to achieve positive optimization. Number of table rows $Rows = 1,000,000$.

Appendix B

Table Rows To Size Conversion

B.1 Mapping From Relative To Absolute Table Size

Throughout our experiments with relational tables, whenever we have spoken of "table size", we have used the number of rows comprising the table to denote *relative* size. The absolute size of a relational table in the Oracle RDBMS is proportional to (*average row size*)×(*number of rows*). The *average row size* in turn is determined by the number of columns and the data types of those columns. Our experimental tables each comprise 20 columns, of which the first five columns are *numeric* and are the targets of our optimization and the remainder are a mix of *string* and *date* data types. Figure B.1 below allows the absolute size of the experimental tables to be determined from the number of rows. The physical space occupied by a table is calculated by adding up the number of bytes occupied of all data segments, including index segments.

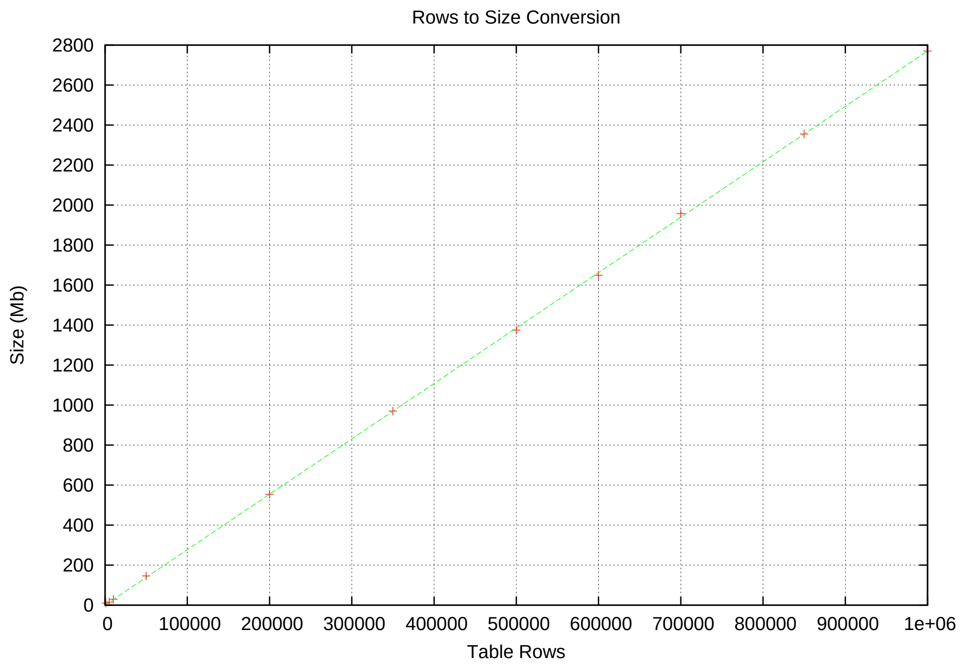


Figure B.1: **Table Rows to Size Conversion**

Appendix C

Software and Hardware Description

C.1 Software Description

The following describes the software employed throughout the experiments reported in this thesis.

C.1.1 Experiments with queries

All experiments were performed using 32 bit Oracle 10.2 running on 32 bit Linux.

Software	**Description**	**Details**	
Operating System	Fedora Core 4	2.6 Linux kernel	32 bit
Database	Oracle 10.2 RDBMS 32 bit		*Bytes*
		Total system global area	1, 174, 405, 120
		Fixed size	1, 219, 040
		Variable size	134, 219, 296
		Database buffers	1, 023, 410, 176
		Redo buffers	15, 556, 608

Table C.1: **Software employed for experiments with queries**.

C.1.2 Experiments with equi-joins

All experiments were performed using 64 bit Oracle 10.2 running on 64 bit Linux.

Software	**Description**	**Details**	
Operating System	Fedora Core 4	2.6 Linux kernel	64 bit
Database	Oracle 10.2 RDBMS 64 bit		*Bytes*
		Total system global area	1, 174, 405, 120
		Fixed size	2, 020, 288
		Variable size	201, 329, 728
		Database buffers	956, 301, 312
		Redo buffers	14, 753, 792

Table C.2: **Software employed for experiments with equi-joins**.

C.2 Hardware Description

The following describes the hardware employed throughout the experiments reported in this thesis.

C.2.1 Experiments with queries

All experiments were performed on a 1.9GHz *AMD Athlon™XP 2600* PC with 2Gb of RAM and standard ATA disk. The Oracle RDBMS was allowed to utilise as much disk space as required.

C.2.2 Experiments with equi-joins

All experiments were performed on a 1.0GHz *AMD Athlon™64 3200* PC with 2Gb of RAM and SATA disk. The Oracle RDBMS was allowed to utilise as much disk space as required.

Bibliography

Aberer, K. & Fischer, G. (1995), Semantic query optimization for methods in object-oriented database systems, *in* ‘ICDE ’95: Proceedings of the Eleventh International Conference on Data Engineering’, IEEE Computer Society, Washington, DC, USA, pp. 70–79.

Agrawal, R., Imieliński, T. & Swami, A. (1993), Mining association rules between sets of items in large databases, *in* ‘SIGMOD ’93: Proceedings of the 1993 ACM SIGMOD international conference on management of data’, ACM Press, New York, NY, USA, pp. 207–216.

Agrawal, S. (2005), ‘Timestamp datatype — Oracle database PL/SQL user’s guide and reference 10g release 2 (10.2)’. [Online; accessed 25-August-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/appdev.102/b14261/datatypes.htm#sthref798*

Albrecht, J., Hümmer, W., Lehner, W. & Schlesinger, L. (2000), Query optimization by using derivability in a data warehouse environment, *in* ‘DOLAP ’00: Proceedings of the 3rd ACM international workshop on Data warehousing and OLAP’, ACM Press, New York, NY, USA, pp. 49–56.

Allen, J. F. (1983), ‘Maintaining knowledge about temporal intervals’, *Commun. ACM* **26**(11), 832–843.

Anton, H. (1984), *Calculus with analytic geometry*, 2 edn, John Wiley and Sons.

Ashdown, L. (2005*a*), ‘Enabling and disabling integrity constraints — Oracle database application developer’s guide - fundamentals, 10g release 2 (10.2)’. [Online; accessed 26-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/appdev.102/b14251/adfns_constraints.htm#i1006697*

Ashdown, L. (2005*b*), ‘Performing date arithmetic — Oracle database application developer’s guide - fundamentals 10g release 2 (10.2)’. [Online; accessed 08-August-2006].

URL: *http://download-west.oracle.com/docs/cd/B19306_01/appdev.102/b14251/adfns_sqltypes.htm#sthref452*

Babcock, B. & Chaudhuri, S. (2005), Towards a robust query optimizer: a principled and practical approach, *in* 'SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data', ACM Press, New York, NY, USA, pp. 119–130.

Bell, S. (1996), Deciding distinctness of query result by discovered constraints, *in* 'Practical Application of Constraint Technology', Practical Application Company, pp. 399–416.

Bloesch, A. C. & Halpin, T. A. (1997), Conceptual queries using conquer-ii, *in* D. Embley & R. Goldstein, eds, 'Proc. ER'97: 16 Int. Conf. on conceptual modeling', Vol. 1331, Springer LNCS, Los Angeles, USA, pp. 113–126.

Broeker, H., Campbell, J., Cunningham, R. & Denholm, D. (2006), 'Gnuplot documentation — gnuplot 4.0'. [Online; accessed 14-September-2006].
URL: *http://www.gnuplot.info/docs/gnuplot.html#fit*

Burleson, D. (1994), *Practical Application of Object-Oriented Techniques to Relational Databases*, 1 edn, John Wiley & Sons Inc.

Chakravarthy, U. S., Grant, J. & Minker, J. (1990), 'Logic-based approach to semantic query optimization', *ACM Trans. Database Syst.* **15**(2), 162–207.

Chamberlin, D. & Boyce, R. (1974), Sequel: A structured english query language, *in* 'SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET Conference', ACM Press, New York, NY, USA.

Chan, I. (2005*a*), 'B-tree indexes — Oracle database performance tuning guide, 10g release 2 (10.2)'. [Online; accessed 14-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/design.htm#sthref140*

Chan, I. (2005*b*), 'Choosing composite indexes — Oracle database performance tuning guide, 10g release 2 (10.2)'. [Online; accessed 23-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/data_acc.htm#sthref1541*

Chan, I. (2005*c*), 'Choosing data block size — Oracle database performance tuning guide, 10g release 2 (10.2)'. [Online; accessed 20-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/iodesign.htm#sthref736*

Chan, I. (2005*d*), 'Managing optimizer statistics — Oracle database performance tuning guide 10g release 2 (10.2)'. [Online; accessed 31-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/stats.htm#PFGRF003*

Chan, I. (2005*e*), 'Shared cursors — Oracle database performance tuning guide, 10g release 2 (10.2)'. [Online; accessed 20-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/memory.htm#i45402*

Chan, I. (2005*f*), 'Using bitmap indexes for performance — Oracle database performance tuning guide, 10g release 2 (10.2)'. [Online; accessed 21-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/data_acc.htm#sthref1587*

Chan, I. (2006*a*), 'Memory configuration and use — Oracle database performance tuning guide 10g release 2 (10.2)'. [Online; accessed 04-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/memory.htm#sthref649*

Chan, I. (2006*b*), 'SQL tuning overview — Oracle database performance tuning guide 10g release 2 (10.2)'. [Online; accessed 06-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/sql_1016.htm#i26072*

Chan, I. (2006*c*), 'Using application tracing tools — Oracle database performance tuning guide 10g release 2 (10.2)'. [Online; accessed 06-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/sqltrace.htm#PFGRF01020*

Chan, I. (2006*d*), 'Using explain plan — Oracle database performance tuning guide 10g release 2 (10.2)'. [Online; accessed 25-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211/ex_plan.htm#PFGRF009*

Chen, I.-M. A. (1996), Query answering using discovered rules, *in* 'ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 402–411.

Cheng, Q., Gryz, J., Koo, F., Leung, T. Y. C., Liu, L., Qian, X. & Schiefer, K. B. (1999), Implementation of two semantic query optimization techniques in DB2 universal database, *in* 'VLDB '99: Proceedings of the 25th International

Conference on Very Large Data Bases', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 687–698.

Cherniack, M. & Zdonik, S. (1998), Changing the rules: transformations for rule-based optimizers, *in* 'SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data', ACM Press, New York, NY, USA, pp. 61–72.

Cherniack, M. & Zdonik, S. B. (1996), Rule languages and internal algebras for rule-based optimizers, *in* 'SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data', ACM Press, New York, NY, USA, pp. 401–412.

Chomicki, J. (2002), Querying with intrinsic preferences, *in* 'EDBT '02: Proceedings of the 8th International Conference on Extending Database Technology', Springer-Verlag, London, UK, pp. 34–51.

Clemmesen, M. (1984), 'Interval arithmetic implementations: using floating point arithmetic', *SIGNUM Newsl.* **19**(4), 2–8.

Cyran, M. & Lane, P. (2003), 'Advantages of b–tree structure — Oracle database concepts, 10g release 1 (10.1)'. [Online; accessed 21-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10743/schema.htm#sthref971*

Cyran, M., Lane, P. & Polk, J. (2005*a*), 'Database buffer cache — Oracle database concepts, 10g release 2 (10.2)'. [Online; accessed 20-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14220/memory.htm#sthref1271*

Cyran, M., Lane, P. & Polk, J. (2005*b*), 'Memory architecture — Oracle database concepts, 10g release 2 (10.2)'. [Online; accessed 31-August-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14220/memory.htm#sthref1257*

Cyran, M., Lane, P. & Polk, J. (2005*c*), 'Overview of bitmap indexes in data warehousing — Oracle database concepts, 10g release 2 (10.2)'. [Online; accessed 22-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14220/bus_intl.htm#sthref2481*

Cyran, M., Lane, P. & Polk, J. (2005*d*), 'Overview of the system global area — Oracle database concepts, 10g release 2 (10.2)'. [Online; accessed

31-August-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14220/memory.htm#sthref1257*

Cyran, M., Lane, P. & Polk, J. (2005*e*), 'Segment space management in locally managed tablespaces — Oracle database concepts, 10g release 2 (10.2)'. [Online; accessed 20-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14220/physical.htm#sthref523*

D'Andrea, A. & Janus, P. (1996), 'UniSQL's next-generation object-relational database management system', *SIGMOD Rec.* **25**(3), 70–76.

Date, C. J. (1995), *An introduction to database systems — Chapter 3*, 6 edn, Addison-Wesley.

Date, C. J. (2003*a*), 'Edgar F. Codd: a tribute and personal memoir', *SIGMOD Rec.* **32**(4), 4–13.

Date, C. J. (2003*b*), *An introduction to database systems — Chapter 18*, 8 edn, Addison-Wesley.

DeHaan, D., Larson, P.-A. & Zhou, J. (2005), Stacked indexed views in microsoft SQL server, *in* 'SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on management of data', ACM Press, New York, NY, USA, pp. 179–190.

Doke, E. R., Satzinger, J. W., Williams, S. R. & Douglas, D. E. (2003), *Object-Oriented Application Development using Visual Basic .NET*, 1 edn, Thomson Course Technology.

Drakos, N. & Moore, R. (2006), 'Gnuplot faq — gnuplot 4.0'. [Online; accessed 14-September-2006].
URL: *http://www.gnuplot.info/faq*

Eisenberg, A. & Melton, J. (2000), 'SQL standardization: the next steps', *SIGMOD Rec.* **29**(1), 63–67.

Fogel, S. & Lane, P. (2006*a*), 'Creating a locally managed tablespace — Oracle database administrator's guide, 10g release 2 (10.2)'. [Online; accessed 20-September-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14231/tspaces.htm#sthref1153*

Fogel, S. & Lane, P. (2006*b*), ‘Developing applications for a distributed database system — Oracle database administrator’s guide 10g release 2 (10.2)’. [Online; accessed 12-July-2006].
URL: *http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14231/ds_appdev.htm#sthref4330*

Gao, D., Jensen, S., Snodgrass, T. & Soo, D. (2005), ‘Join operations in temporal databases’, *The VLDB Journal* **14**(1), 2–29.

Gemignani, M. C. (1990), *Elementary Topology*, 2 edn, Courier Dover Publications.

Genet, B. & Dobbie, G. (1998), Is semantic optimisation worthwhile?, *in* ‘Proceedings of the 21st Australasian Computer Science Conference’.

Godfrey, P., Grant, J., Gryz, J. & Minker, J. (1998), Integrity constraints: semantics and applications, *in* ‘Logics for databases and information systems’, Kluwer Academic Publishers, Norwell, MA, USA, pp. 265–306.

Godfrey, P. & Gryz, J. (1996), A framework for intensional query optimization, *in* D. Boulanger, U. Geske, F. Giannotti & D. Seipel, eds, ‘DDLP ’96: Workshop on Deductive Databases and Logic Programming’, pp. 57–68.

Godfrey, P., Gryz, J. & Minker, J. (1996), Semantic query optimization for bottom-up evaluation, *in* Z. W. Rás & M. Michalewicz, eds, ‘Proceedings of the Ninth International Symposium on Foundations of Intelligent Systems’, Vol. 1079 of *LNAI*, Berlin, pp. 561–571.

Godfrey, P., Gryz, J. & Zuzarte, C. (2001), Exploiting constraint-like data characterizations in query optimization, *in* ‘SIGMOD ’01: Proceedings of the 2001 ACM SIGMOD international conference on management of data’, ACM Press, New York, NY, USA, pp. 582–592.

Grant, J., Gryz, J., Minker, J. & Raschid, L. (1997), Semantic query optimization for object databases, *in* ‘ICDE ’97: Thirteenth International Conference on Data Engineering’, IEEE Computer Science Press, Los Amitos, California, USA, pp. 444–453.

Gryz, J., Liu, L. & Qian, X. (1999), Semantic query optimization in IBM DB2: Initial results, Technical Report CS-1999-01, Department of Computer Science, York University, 4700 Keele Street, North York, Ontario M3J 1P3, Canada.

Gryz, J., Schiefer, K. B., Zheng, J. & Zuzarte, C. (2001), Discovery and application of check constraints in DB2, *in* ‘Proceedings of the 17th International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 551–556.

Haas, L. M., Carey, M. J., Livny, M. & Shukla, A. (1997), 'Sing the truth about ad hoc join costs', *The VLDB Journal* **6**(3), 241–256.

Hammer, M. & Zdonik, S. B. (1980), Knowledge based query processing, *in* 'VLDB '80: Proceedings of the 6th International Conference Very Large Data Bases', Morgan-Kaufman, pp. 137–147.

Han, J., Cai, Y. & Cercone, N. (1993), 'Data-driven discovery of quantitative rules in relational databases', *IEEE Transactions on Knowledge and Data Engineering* **5**(1), 29–40.

Han, J. W., Huang, Y., Cercone, N. & Fu, Y. J. (1996), 'Intelligent query answering by knowledge discovery techniques', *IEEE Transactions on Knowledge and Data Engineering* **8**, 373–390.

Hickey, T., Ju, Q. & Emden, M. H. V. (2001), 'Interval arithmetic: From principles to implementation', *J. ACM* **48**(5), 1038–1068.

Hobbs, L., Hillson, S., Lawande, S. & Smith, P. (2004), *Oracle 10g Data Warehousing*, 1 edn, Digital Press.

Hsu, C. & Knoblock, C. A. (2000), 'Semantic query optimization for query plans of heterogeneous multidatabase systems', *IEEE Transactions on Knowledge and Data Engineering* **12**(6), 959–978.

Hsu, C.-N. & Knoblock, C. A. (1994), Rule induction for semantic query optimization, *in* 'ML '94: Proc. 11th International Conference on Machine Learning', Morgan Kaufmann, pp. 112–120.

Hsu, C.-N. & Knoblock, C. A. (1996), Using inductive learning to generate rules for semantic query optimization, *in* G. Piatetsky-Shapiro, U. Fayyad, P. Symyth & R. Uthurusamy, eds, 'Advances in knowledge discovery and data mining', American Association for Artificial Intelligence, Menlo Park, CA, USA, pp. 425–445.

Hsu, C.-N. & Knoblock, C. A. (1998), 'Discovering robust knowledge from databases that change', *Data Min. Knowl. Discov.* **2**(1), 69–95.

IEEE (1985), IEEE standard for binary floating point arithmetic, Technical Report 754-1985, IEEE Standards Board, Los Alamitos, California, USA.

Illarramendi, A., Blanco, J. M. & Goni, A. (1994), 'Making knowledge base systems more efficient: a method to detect inconsistent queries', *IEEE Transactions on Knowledge and Data Engineering* **6**(4), 634–639.

Jarke, M. & Koch, J. (1984), 'Query optimization in database systems', *ACM Computing Surveys* **16**(2), 111–152.

Kim, S.-K. & Chakravarthy, S. (1992), A retrospective analysis of time concepts in temporal databases, Technical Report UF-CIS-TR-92-044, University of Florida, FL, USA.

King, J. J. (1981), QUIST: A system for semantic query optimization in relational databases, *in* 'VLDB '81: Proceedings of the 7th International Conference on Very Large Databases', IEEE Computer Society Press, pp. 510–517.

Kriegel, H.-P., Pötke, M. & Seidl, T. (2001), Object-relational indexing for general interval relationships, *in* 'SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases', Springer-Verlag, London, UK, pp. 522–542.

Krokhin, A., Jeavons, P. & Jonsson, P. (2003), 'Reasoning about temporal relations: The tractable sub-algebras of Allen's interval algebra', *J. ACM* **50**(5), 591–640.

Lane, P. & Schupmann, V. (2002), 'Overview of extraction, transformation, and loading — Oracle9i data warehousing guide, release 2 (9.2)'. [Online; accessed 25-September-2006].
URL: *http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96520/ettoverv.htm#1020*

Larsen, R. & Marx, M. (1981), *An Introduction to Mathematical Statistics and its Applications*, 1 edn, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, USA.

Lee, M. L., Bressan, S., Goh, C. H. & Ramakrishnan, R. (1999), Integration of disparate information sources: A short survey, *in* 'Workshop on Logic Programming and Distributed Knowledge Management', UK.

Lowden, B. G. T. & Robinson, J. (1999), A statistical approach to rule selection in semantic query optimisation, *in* 'ISMIS '99: Proceedings of the 11th International Symposium on Foundations of Intelligent Systems', Springer-Verlag, London, UK, pp. 330–339.

Lowden, B. G. T. & Robinson, J. (2002), Constructing inter-relational rules for semantic query optimisation, *in* 'DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications', Springer-Verlag, London, UK, pp. 587–596.

Lowden, B. G. T. & Robinson, J. (2004), Improved data retrieval using semantic transformation, *in* 'DEXA '04 : Proceedings of the 15th International Conference on Database and Expert Systems Applications', Springer, Berlin, Germany, pp. 391–400.

Luscher, L. & Green, C. D. (2002), 'Using the rule-based optimizer — Oracle9i database performance tuning guide and reference release 2 (9.2)'. [Online; accessed 21-September-2006].
URL: *http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/toc.htm*

Maiocchi, R., Pernici, B. & Barbic, F. (1992), 'Automatic deduction of temporal information', *ACM Trans. Database Syst.* **17**(4), 647–688.

Mani, I., Pustejovsky, J. & Sundheim, B. (2004), 'Introduction to the special issue on temporal information processing', *ACM Transactions on Asian Language Information Processing (TALIP)* **3**(1), 1–10.

Mannila, H., Toivonen, H. & Verkamo, A. I. (1994), Efficient algorithms for discovering association rules, *in* 'KDD-94: AAAI Workshop on Knowledge Discovery in Databases', pp. 181–192.

Maurer, S. B. (2004), *Discrete Algorithmic Mathematics*, 1 edn, A K Peters, Ltd.

Miller, R. J. & Yang, Y. (1997), Association rules over interval data, *in* 'SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on management of data', ACM Press, New York, NY, USA, pp. 452–461.

Muñoz, C. & Lester, D. (2005), Real number calculations and theorem proving, *in* J. Hurd & T. Melham, eds, 'TPHOLs '05: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics', Vol. 3603, Springer-Verlag, Oxford, UK, pp. 195–210.

Nebel, B. & Bürckert, H.-J. (1995), 'Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra', *J. ACM* **42**(1), 43–66.

Özsoyoglu, G. & Snodgrass, R. T. (1995), 'Temporal and real-time databases: A survey', *IEEE Transactions on Knowledge and Data Engineering* **7**(4), 513–532.

Pang, H. H., Lu, H. J. & Ooi, B. C. (1991), An efficient semantic query optimization algorithm, *in* 'ICDE '91: Proceedings of the IEEE International Conference on Data Engineering', IEEE Computer Society Press, Los Alamitos, Ca., USA, pp. 326–335.

Park, J. S., Chen, M.-S. & Yu, P. S. (1995), An effective hash-based algorithm for mining association rules, *in* 'SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on management of data', ACM Press, New York, NY, USA, pp. 175–186.

Piatetsky-Shapiro, G. (1991), Discovery, analysis and presentation of strong rules, *in* G. Piatetsky-Shapiro & W. Frawley, eds, 'Knowledge Discovery in Databases', AAAI/MIT Press, pp. 229–248.

Pohl, I. & Shaw, A. (1986), *The Nature of Computation: An Introduction to Computer Science*, 6 edn, Computer Science Press Inc, Rockville, Maryland 20850, USA.

Poole, D. (2005), *Linear Algebra - A Modern Introduction*, 2 edn, Thomson Brookes/Cole.

Press, W., Flannery, B., Teukolsky, S. & Vetterling, W. (1992), *Numerical Recipes in C: The Art of Scientific Computing*, 2 edn, Cambridge University Press, Cambridge, UK.

Rich, K. (2005), 'Uses of initialization parameters — Oracle database reference, 10g release 2 (10.2)'. [Online; accessed 31-August-2006].
URL: `http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14237/initparams001.htm#i1124342`

Rishe, N., Sun, W. & Barton, D. (1995), 'Florida international university high performance database research center', *SIGMOD Rec.* **24**(3), 71–76.

Rishe, N., Sun, W. & Barton, D. (2003), 'Mining for empty spaces in large data sets', *Theoretical Computer Science* **296**(3), 435–452.

Savasere, A., Omiecinski, E. & Navathe, S. B. (1995), An efficient algorithm for mining association rules in large databases, *in* 'VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 432–444.

Sayli, A. & Lowden, B. (1996), The use of statistics in semantic query optimisation, *in* 'Proc. 13th. European Meeting on Cybernetics and Systems Research', pp. 991–996.

Sciore, E. & Siegel, M. (1990), Heuristic-based semantic query optimization, *in* ‘JCIT ’90: Proceedings of the fifth Jerusalem conference on information technology’, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 541–550.

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

Shekhar, S., Srivastava, J. & Dutta, S. (1992), ‘A formal model of trade-off between optimization and execution costs in semantic query optimization’, *Data and Knowledge Engineering* **8**(2), 131–151.

Shenoy, S. T. & Ozsoyoglu, Z. M. (1987), A system for semantic query optimization, *in* ‘SIGMOD ’87: Proceedings of the 1987 ACM SIGMOD international conference on management of data’, ACM Press, New York, USA, pp. 181–195.

Shenoy, S. T. & Ozsoyoglu, Z. M. (1989), ‘Design and implementation of a semantic query optimizer’, *IEEE Transactions on Knowledge and Data Engineering* **1**(3), 344–361.

Siegel, M., Sciore, E. & Salveter, S. (1992), ‘A method for automatic rule derivation to support semantic query optimisation’, *ACM Transactions on Database Systems* **17**(4), 563–600.

Snodgrass, R. & Ahn, I. (1985), A taxonomy of time databases, *in* ‘SIGMOD ’85: Proceedings of the 1985 ACM SIGMOD international conference on management of data’, ACM Press, New York, USA, pp. 236–246.

Srikant, R. & Agrawal, R. (1996), Mining quantitative association rules in large relational tables, *in* ‘SIGMOD ’96: Proceedings of the 1996 ACM SIGMOD international conference on management of data’, ACM Press, New York, NY, USA, pp. 1–12.

Sun, W. & Yu, C. (1994), ‘Semantic query optimization for tree and chain queries’, *IEEE Transactions on Knowledge and Data Engineering* **6**(1), 136–151.

Waas, F. & Galindo-Legaria, C. (2000), Counting, enumerating, and sampling of execution plans in a cost-based query optimizer, *in* ‘SIGMOD ’00: Proceedings of the 2000 ACM SIGMOD international conference on management of data’, ACM Press, New York, NY, USA, pp. 499–509.

Walster, G. W. (2000), 'The use and implementation of interval data types in Fortran', *SIGPLAN Fortran Forum* **19**(2), 2–15.

Warshaw, L. B. & Miranker, D. P. (1999), Rule-based query optimization, revisited, *in* 'CIKM '99: Proceedings of the eighth international conference on Information and knowledge management', ACM Press, New York, NY, USA, pp. 267–275.

Yoon, S., Henschen, L. J., Park, E. K. & Makki, S. (1999), Using domain knowledge in knowledge discovery, *in* 'CIKM '99: Proceedings of the eighth international conference on information and knowledge management', ACM Press, New York, USA, pp. 243–250.

Yu, C. T. & Sun, W. (1989), 'Automatic knowledge acquisition and maintenance for semantic query optimization', *IEEE Transactions on Knowledge and Data Engineering* **1**(3), 362–375.

Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H. & Urata, M. (2000), Answering complex SQL queries using automatic summary tables, *in* 'SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on management of data', ACM Press, New York, NY, USA, pp. 105–116.

Zhang, X. & Ozsoyoglu, Z. M. (1997), 'Implication and referential constraints: A new formal reasoning', *IEEE Transactions on Knowledge and Data Engineering* **9**(6), 894–910.

Zhu, Q. (1992), Query optimization in multidatabase systems, *in* 'CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research', IBM Press, pp. 111–127.