INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Semantic query processing in database systems

Shenoy, Sreekumar Thrivikrama, Ph.D.
Case Western Reserve University, 1990
SEMANTIC QUERY PROCESSING IN DATABASE SYSTEMS

by

SREEKUMAR THRIVIKRAMA SHENOY

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Computer Engineering and Science Department

CASE WESTERN RESERVE UNIVERSITY

January 1990
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

SREEKUMAR T. SHENOY

candidate for the Ph.D. degree.*

Signed:  

[Signatures]

(Chairman)

Date 17th NOVEMBER, 1989

*We also certify that written approval has been obtained for any proprietary material contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

T. Sree Kumar
7th Dec, '89
SEMANTIC QUERY PROCESSING IN DATABASE SYSTEMS

ABSTRACT

by

SREEKUMAR THRIVIKRAMA SHENOY

In this thesis we describe a scheme to represent, utilize, and maintain semantic knowledge in optimizing user specified queries. The semantics is represented as function-free clauses in predicate logic. Knowledge is formulated as semantic constraints pertaining to the database, and is assumed to be satisfied by all the instances of the database. We use three distinct types of semantic constraints, namely, implication constraints, subset constraints, and aggregate constraints.

The proposed scheme uses a graph theoretic approach to identify redundant joins and restrictions present in a given query. An optimization algorithm is presented which eliminates semantically redundant non-profitable specifications from a query while adding redundant profitable specifications to it. The optimization algorithm is guided by a set of heuristics and utilizes dynamic interaction of three entities - schema, semantics, and query - for semantic query transformation. Various sequential stages of the algorithm are described that deal with semantic expansion, relation elimination, restriction elimination etc. The complexity of the algorithm and cost reduction of semantic optimization are analyzed in detail.
The implementation architecture of the algorithm and test results on a representative set of data are presented. Details on all the associated user interfaces are described. The test results reveal the potential advantages of a semantic optimizer in conjunction with a conventional one.

Many potential inconsistencies possible with a growing set of semantic constraints are identified. Maintenance issues associated with different types of constraints are addressed. Algorithms for semantic maintenance regarding redundancy and contradiction are introduced for subset, implication, and aggregate constraints.
To my parents, S. Thrivikrama Shenoy and K.N. Leela Bai.

To my wife Rani and daughter Rathna.

And to the memory of my grand mother and aunt,

who did not live to share my joy of graduation.
ACKNOWLEDGEMENTS

I am literally indebted to many individuals and institutions for their help and encouragement during my six years of this thesis work. I am particularly thankful to my thesis advisor and guide Prof. Zehra Meral Ozsoyoglu for her guidance, patience, and tolerance. I acknowledge and thank Prof. Tekin Ozsoyoglu for many helpful discussions we have had.

I am thankful to the HSEQ department of B.P America for its continuous and generous support for more than five years now. I am sincerely grateful to my manager Dr. David P. Gillespie of B.P America for his unlimited moral support. I thank my colleague Ms. Shampra Gay of B.P America for sharing my work load all along these critical years.

I acknowledge the encouragement, support and understanding of my wife Rani and daughter Rathna, without which this work would not have been possible.
# TABLE OF CONTENTS

Abstract ii
Dedication
Acknowledgement vi
Table of contents
List of figures ix

1. Introduction 1

2. Previous work 6
   2.1 Overview 6
   2.2 Limitations 8

3. Semantic constraints 10
   3.1 Clausal representation 10
   3.2 Types of semantic constraints 12
   3.3 Subset constraints 13
   3.4 Implication constraints 14
   3.5 Aggregate constraints 17
   3.6 Example database and constraints 19

4. Constraint utilization 22
   4.1 Heuristics and inference rules 22
   4.2 Restriction elimination 22
   4.3 Index introduction 23
   4.4 Scan reduction 23
   4.5 Join elimination 24
8. Constraint maintenance

8.1 Maintenance of subset constraints
   8.1.1 Maintenance task
   8.1.2 Assumption
   8.1.3 Data structures
   8.1.4 Algorithm for constraint maintenance
   8.1.5 Complexity of the algorithm

8.2 Maintenance of implication constraints
   8.2.1 Deduction system for new constraints
   8.2.2 Intra-constraint redundancy/contradiction
   8.2.3 Inter-constraint redundancy/contradiction
   8.2.4 Relevant subsets of a constraint
   8.2.5 Constraint derivation from equivalence classes

8.3 Maintenance of aggregate constraints
   8.3.1 Interaction of simple and aggregate constraints
   8.3.2 Generalization of derivation
   8.3.3 Maintenance of aggregate constraints

9. Conclusion and future extensions

References
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig 5.1</td>
<td>Query graph, $G_q$</td>
<td>29</td>
</tr>
<tr>
<td>Fig 5.2</td>
<td>Canonical graph, $G_c$</td>
<td>35</td>
</tr>
<tr>
<td>Fig 5.3</td>
<td>Semantic Expansion</td>
<td>38</td>
</tr>
<tr>
<td>Fig 5.4</td>
<td>Relation elimination, $G_{rl}$</td>
<td>41</td>
</tr>
<tr>
<td>Fig 5.5</td>
<td>Converted form, $G_f$</td>
<td>45</td>
</tr>
<tr>
<td>Fig 7.1</td>
<td>Specification module</td>
<td>56</td>
</tr>
<tr>
<td>Fig 7.2</td>
<td>Maintenance module</td>
<td>57</td>
</tr>
<tr>
<td>Fig 7.3</td>
<td>Test results of semantic optimization</td>
<td>64</td>
</tr>
<tr>
<td>Fig 8.1</td>
<td>A conjunction graph</td>
<td>100</td>
</tr>
<tr>
<td>Fig 8.2</td>
<td>Derivation of $op_m$, from $Agg$ and $op_m$</td>
<td>108</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Query optimization in relational databases has been an active issue in both academic and commercial fields for a quite long time now. The relevance for optimization stems from the flexibility provided by modern user-interfaces to databases. The interfaces and non-procedural query languages enable the users to specify queries which may be computationally costly and inefficient to process. It then becomes important to reformulate the user specified query to an equivalent form that is computationally more efficient.

Query optimization can be formally defined as a process of transforming a query into an equivalent form (that produces the same result as the original one for all database states) which can be evaluated more efficiently.

Optimization in its conventional sense utilizes syntactic knowledge of the operations and storage details of the relations. The syntactic knowledge includes algebraic transformations and operator resequencing, whereas the storage details include indices and clustering of storage. Several query processing algorithms are proposed in the literature [Aho79, Astr76, Bern81, Blas77,
Epst78, Gran80, Gotl75, Hevn79, Kim79, KrBZ86, Maie83, Pale74, Sell86, Ston76, Ullm82, Wong76, Yao79]. Most of the major commercial database management systems utilize these techniques to some extent in answering ad hoc user queries.

Semantic processing adds a relatively new dimension to query optimization. Instead of just resequencing the operators or incorporating the indexed access of data files, it tries to exploit any available knowledge about the data. For instance, it utilizes knowledge about the domains of relations, nature of data, and constraints associated with database instances. Such relevant pieces of knowledge available to the optimizer, combined with its potential ability to intelligently process it, helps it to generate forms of the user specified query which are better from an execution point of view.

Significance of semantic optimization can be made more apparent by certain inherent limitations of syntactic optimization techniques. Since syntactic optimizers lack the entire body of semantic knowledge assured to be satisfied by all the instances of a particular database, in many cases they produce suboptimal forms of the query for execution. Certain queries that can be answered without any relation scans cannot be detected by syntactic optimizers, thus resulting in redundant database access. Cases
where queries contain dangling relations cannot be identified by syntactic techniques alone, thus forcing redundant joins to be performed. Also, syntactic optimizers cannot detect and eliminate semantically redundant restrictions or joins from user specified queries, and for the same reason they fail to introduce semantically redundant restrictions or joins which could, in turn, reduce the overall cost of the query.

Semantic query optimization is based on the semantic equivalence rather than the syntactic equivalence between different queries. Two queries are syntactically equivalent if their answers are the same for all the instances of the database. Two queries, possibly syntactically non-equivalent, are semantically equivalent if their answers are the same for all the instances of the database that satisfy the specified set of semantic rules. Semantic equivalence does not imply syntactic equivalence while syntactic equivalence trivially implies semantic one. As an example, the two queries "retrieve (emp.all) where emp.Sal>40K" and "retrieve (emp.all) where emp.Sal>40K and emp.Job='Manager'" are not syntactically equivalent, but are semantically equivalent under the semantic rule 'emp.Sal>40K \rightarrow emp.Job='Manager'". Since the semantic equivalence between queries depends only on the database schema and the semantic rule set, the different queries can interchangeably be used to get the
same results, provided the schema and the semantics are unaltered. Moreover, since these syntactically non-equivalent queries can independently be optimized by a conventional syntactic optimizer, semantic processing does in fact expand the spectrum of equivalent forms of the specified query. Thus, the semantic query optimization is the process of finalizing, among all the possible syntactically and semantically equivalent forms of the query, the one which can be executed most efficiently.

There are various issues involved in semantic query processing. Firstly, query and schema should be dynamically used to select the relevant semantics for optimization without an exhaustive search of semantic rule base. Secondly, there should be some mechanism to merge the selected semantics with the query. Thirdly, there should be a cost analyzer to evaluate the costs of equivalent queries and rank them accordingly. Fourthly, there should be a set of heuristics to guide the whole process in a meaningful way without a combinatorial explosion.

This work is organized as follows. Chapter 2 presents a brief discussion on the related previous work. In chapter 3 we discuss clausal representations of query as well as various types of constraints that constitute a semantic rule base for the optimizer. We introduce a simplified and generalized
representation of implication constraints. Chapter 4 addresses the issues related with the maintenance of semantic constraints. A maintenance algorithm is presented in this chapter with its associated data structures. Chapter 5 illustrates the role of heuristics as inference rules. Different graph schemes used to represent and transform the query are introduced in chapter 6. A detailed discussion on various stages of semantic query transformation appears in chapter 7. In chapter 8 we present the transformation algorithm, its implementation architecture, and the implementation results. Chapter 9 concludes the work.
Chapter 2

PREVIOUS WORK

2.1 Overview

Semantic optimization has recently been the subject of detailed analysis from two different perspectives. [King81] formally introduced the issue in an Artificial Intelligence context and introduced a set of heuristics for query transformation. [Hazd80] analyzed the problem in a database point of view.

Major heuristics discussed in [King81] were index introduction, join introduction, scan reduction, and join elimination. Index introduction tries to obtain a constraint on an attribute of a relation which is restricted in the query and which has a clustered indexed attribute that is not restricted in the query. According to the strategy of join introduction, a relation should be a constraint target if it has a clustering link into a much larger relation that is constrained in the query, even if the relation itself is not in the original query. This heuristic contemplates addition of a join to the query, referred to as join introduction. In the case of scan reduction, the objective is to reduce the number of inner scans of the join by
obtaining additional restrictions prior to the cross referencing part of the operation. Join elimination becomes possible if a relation is joined to just one other relation and none of its attributes contribute to the answer.

Later, a substantial amount of research followed, related to the theory and implementation of semantic rules for query processing [ChFM84, ChMi85, ChMG86, Jark84, JaCV84, Morg84, Sieg87, Sieg88].

Two of the above papers, [ChFM84] and [Jark84], have maximum relevance to our current work. [ChFM84] introduces the concept of semantic compilation, where all the relevant semantic rules are explicitly associated with each relations or view definitions. This allows any query on that relation or view to be semantically transformed with only a limited search of the rule base. The result of interaction of a query with compiled relations or views is a group of semantically equivalent queries, each of which can be potentially optimized using a syntactic optimizer. [Jark84] describes a graph theoretic approach integrated with tableau techniques and syntactic simplification algorithms to optimize queries containing inequality constraints. Referential integrity constraints like key dependencies, functional dependencies, and value bounds are used by the algorithm. The graph is used to
unify attribute values based on referential constraints, to detect cycles that imply equal values for different attributes, and to predict queries with null answers.

2.2 Limitations

Both of the above methods have certain limitations. [ChFM84] fails to clearly categorize a given piece of semantic information as a rule or as a view. Also, no method is available to select or prioritize the rules associated with a relation or view in a query context. Moreover, no mechanism is available to quantify the profitability of a rule for a relation in a query context. In other words, integration of semantic rules with relations is considered in isolation with query context. In [Jark85], explicit representation of arbitrary semantic rules is not supported. Prolog like view characterization is used to express a limited type of constraints on the variables appearing in view definitions. Since semantic details are integrated with view definitions, it becomes the responsibility of the end user to keep track of the semantics associated with each view. Since the constraints are hardwired to view definitions, they become unsharable by the similar attributes originating from the query. Also, any changes in the constraints at a later stage makes the maintenance of these views difficult.
In [ShOz87] and [ShOz89] we try to address the above difficulties by using explicit clausal representation [Kowa83] of integrity constraints as in [ChFM84], and by devising a mechanism for dynamic interaction between relations and constraints in a query context. Among the valid constraints selected for such interaction only the profitable ones are finally used, profitability being decided by heuristic rules, global parameters, and some assumptions.
Chapter 3

SEMANTIC CONSTRAINTS

In this section we discuss the clausal representation [Kowal83, StSh86, Ullm88] which provides a theoretical basis for specifying semantic constraints. After introducing the notations of clausal form, we describe the specific representational details of the constraints.

3.1 Clausal representation

A clause is an expression of the form \( A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m \) where \( A_1, \ldots, A_n, B_1, \ldots, B_m \) are atoms (or atomic formulae), \( n \geq 0 \) and \( m \geq 0 \). An atom (or atomic formula) is an expression of the form \( p(t_1, \ldots, t_j) \) where \( p \) is a \( j \)-place predicate symbol, \( t_1, \ldots, t_j \) are terms, and \( j \geq 1 \). A term, in its most general form, is a variable, a constant symbol, or a function \( f(t_1, \ldots, t_k) \) where \( f \) is a \( k \)-place function symbol, \( t_1, \ldots, t_k \) are terms and \( k > 0 \).

In our discussion we consider only function-free terms. A function-free atomic formula, \( p(t_1, \ldots, t_j) \), denotes either a relation or a built-in predicate. If it is a relation, it is the relation of its predicate. Relational predicates may be
restricted for equality by any constant appearing in a component or may be equi-joined between components that have the same variable. If it is a built-in predicate, it is a binary comparison (arithmetic or set) predicate of the form =, ≠, >, ≥, "contains" etc. We follow the usual infix notation X > Y, instead of >(X,Y), to represent the built-in predicates.

A literal is either an atomic formula or a negated atomic formula. A non-negated atomic formula is positive literal, and a negated one is a negative literal. In the above clause, A₁,...,Aₙ are the negative literals and are referred to as the body of the clause, whereas B₁,...,Bₘ are the positive literals and represent its head. The atoms appearing in the body are the joint (conjunctive) conditions of the clause, and the ones in its head are the alternative (disjunctive) conclusions. The conditions are sometimes referred to as antecedent atoms and the conclusions as consequent atoms.

A clause, thus, is a sum (logical OR) of literals. A clause with at most one positive literal is called a Horn clause, which can be of one of the following categories:

1. Integrity constraint: No positive literal, one or more negative literals (m=0, n>0).
2. **Unit clause or Fact:** A single positive literal, no negative literals \((m=1,n=0)\).

3. **Rule:** A single positive literal, one or more negative literals \((m=1,n>0)\).

### 3.2 Types Semantic constraints

Constraints are laws or expressions associated with the database that represent certain required properties of the data. There are two broad classifications of constraints, i.e., state constraints and transition constraints [NiYa78]. In this paper, we restrict our discussion to state constraints. The state constraints can be further classified into conventional dependencies and semantic constraints. Conventional dependencies include functional (and key) dependencies, value bounds, referential constraints etc. A detailed discussion can be found in [Ullm82]. Semantic constraints represent inter-relationships between chunks of data across the database relations.

In this work, we utilize three types of semantic constraints, viz, **subset constraints** \(S\), **implication constraints** \(I\), and **aggregate constraints** \(A\), defined over the database scheme \(D\). In other words our **complete semantic specification** has four components, \(D,S,I\), and \(A\). Clausal forms are used to represent the
semantic constraints. All the three types of the semantic constraints of our interest can be represented by integrity constraints (conjunctions of negated predicates). In other words, we do not consider the other variants of horn clauses ("facts" or "rules") for constraint specification. Usually, the semantic integrity constraints contain relational predicates as well as built-in predicates.

3.3 Subset constraints

**Definition 3.1:** The set of subset constraints $S$ is a superset-subset relationship between the domains of two different attributes of possibly two different relations.

A subset constraint is represented by an integrity constraint (a conjunction of negated predicates) having two relational predicates and one built-in predicate. The built-in predicate specifies a set comparison between two attributes of the relations. Note that no restriction (constant substitution) is allowed on any attribute variables of the relational predicates.

An example of a subset constraint is "$r_1(X_1, Y_1, Z_1), r_2(X_2, Y_2, Z_2), X_1 \subseteq X_2 \rightarrow \$", which is same as "$r_1.X_1 \subseteq r_2.X_2 \rightarrow \$" if we decide to prefix the attributes by the relation names. This
constraint states that the condition "r₁,X₁ ⊆ r₂,X₂" is always false. In other words, it restricts the domain of r₂,X₂ to be a subset of the domain of r₁,X₁. If the built-in predicate is complemented and moved to the head, the specification becomes "→ r₂,X₂ ⊆ r₁,X₁", equivalent to saying that domain of r₁,X₁ is a superset of that of r₂,X₂. Note that, as in a query, shared variables can be used in a subset constraint to specify equality implicitly.

A classic example of subset constraint is: "all managers are employees".

3.4 Implication constraints

Formally, an implication constraint is represented by a clausal integrity constraint, (a horn clause with no positive literal, and one or more negative literals), which is a conjunction of negated predicates. In the most basic form, these predicates could be relational predicates or built-in predicates. The relational predicates represent the database relations (or views), whereas the built-in predicates represent comparison between a variable and a constant or comparison between two variables [RoHu80].
Implication constraints restrict the relative domains of attributes. They specify valid ranges of values that certain attributes can have when some other attributes are restricted in the same or a different relation.

As an example, the constraint "Only managers make more than 40K" on the employee relation can be represented as:

\[
\text{employee(Ssn, Name, Dept, Job, Grade, Sal, Bonus, Age), } \text{Sal > 40K} \rightarrow \text{Job = "Manager"}
\]

Here, "employee(...)" is the relational predicate and the other two are the built-in ones. In our discussion, we eliminate the explicit representation of relational predicates and prefix the attributes with relation names for improving the readability without losing any generality. The above example with such a representation would be:

\[
\text{employee.Sal > 40K} \rightarrow \text{employee.Job = "Manager".}
\]

Here we introduce a simple generalization to the above representation. We complement and move the "consequent" predicate (employee.Job = "Manager") to the "antecedent" side, thus making
it a "true" clausal integrity constraint (with no positive literal). The resultant representation in our example is:

\[ \text{employee.Sal} > 40K, \text{employee.Job} \neq "Manager" \rightarrow . \]

The constraint can now be read as 'there is no tuple in the employee relation with Sal > 40K and Job \neq "Manager"'. There are two definite advantages to this modification. Firstly, it simplifies the syntax of the constraint by associating it to a simple conjunctive set of predicates, thus eliminating the classification of "antecedent-consequent" atoms. Secondly, it generalizes the semantics of the constraint, as any of the predicates in the conjunctive set qualifies to be the consequent one when complemented and moved to the "consequent" side. For example, from the above conjunctive set, we can also derive:

\[ \text{employee.Job} \neq "Manager" \rightarrow \text{employee.Sal} \leq 40K \]

Semantically, an implication constraint represents an impossible conjunctive combination. From the database point of view, the conjunction represented by an implication constraint always evaluates to be false.
An implication constraint is said to be local if all its relational predicates refer to the same relation. Otherwise it is called a cross constraint because, in such cases, the implication relates more than one relation. The cross implication constraints involve at least one join specification between relations.

3.5 Aggregate constraints

In this section we introduce aggregate functions in implication integrity constraints. We refer to an implication constraint containing no aggregate functions as a simple implication constraint, and the one involving aggregate functions as an aggregate implication constraint.

Semantically, an aggregate implication constraint can be considered as first defining a view by a set of qualification clauses and then specifying an aggregate attribute value on the defined view by an aggregation clause. The set of qualifications can contain restrictions as well as join clauses, thus making it possible to specify aggregate constraints on single relations as well as on multiple ones connected by join clauses.

As an example, consider the aggregate constraint: "there are 140 employees who make more than 45K". This can be expressed by
first forming a view of all the employees who make more than 45K, and then specifying a count aggregate on this view to be 140. We represent this by "\( \sigma(\text{employee.Sal} > 45K) \mid \text{count(employee.Ssn)} = 140 \)", where "Ssn" is assumed to the relation key. Similarly the aggregate constraint "minimum age of a manager is 45" can be represented as "\( \sigma(\text{employee.Job} = 'Manager') \mid \text{min(employee.Age)} = 45 \)".

An aggregate implication constraint is of the general form:
\[
\sigma(r.A_1 \text{ op}_1 k_1, r.A_2 \text{ op}_2 k_2, \ldots, r.A_n \text{ op}_n k_n) | \text{Agg}(r.A_m) \text{ op}_m k_m,
\]
where \( r \) represents the relation; \( A_1, \ldots, A_n \) and \( A_m \) represent the attributes; \( \text{op}_1, \ldots, \text{op}_n \) and \( \text{op}_m \) represent the comparison operators from the set \( \{=,\neq,>,\geq,<,\leq\} \); and \( k_1, k_2, \ldots, k_n \) and \( k_m \) represent constants. The domain of the attribute \( A_m \) is assumed to have only positive values. This restriction will be relaxed in certain cases later. For a conjunction of predicates \( P \), \( \sigma(P) \) denotes the set of tuples satisfying the conjunction. \( \text{Agg} \) represents an aggregate function from the set \( \{\text{min}, \text{max}, \text{sum}(u), \text{avg}(u), \text{count}(u)\} \), representing minimum, maximum, sum(unique), average(unique), and count(unique) respectively. The aggregation symbol "|" specifies an aggregate value on its right hand side for the set of tuples satisfying the qualifications on its left hand side. For notational convenience, we write the above constraint by: \( \sigma(\Pi_i r.A_i \text{ op}_i k_i) | \text{Agg}(r.A_m) \text{ op}_m k_m \). This constraint states
that for the set of tuples satisfying the conjunction of qualification predicates on the left hand side, \( A_1 \ op_1 \ k_1, \ldots \)
the aggregate specification on the right hand side, \( \text{Agg}(r.A_m) \ op_m k_m \), holds.

3.6 Example database and constraints

We use the following schema and constraints for various illustrations in this paper:

\textbf{Schema:} (Indexed attributes are underlined).

employee (\textit{Ssn}, Name, Dept, \textit{Job}, Grade, Sal, Bonus, Age)
storage (\textit{Dept}, Material, Qty)
material (\textit{Material}, Risk, Storage\_Limit)
dept (\textit{Dept}, Location)

\textbf{Subset constraints:}

SC1: Materials should be first present in "material" table before "storage".

storage.Material \subseteq material.Material
Implication constraints, implicative and conjunctive forms:

IC₁: Only managers make more than 40K.

employee.Sal > 40K → employee.Job = "Manager".


IC₂: All managers are of grade 20 or higher.


IC₃: All materials stored in department d₁ are of risk greater than 3.


IC₄: Benzene is always stored in quantities more than 500.

storage.Material = "Benzene" → storage Qty > 500.

storage.Material = "Benzene", storage Qty ≤ 500 →.

IC₅: Employees of any department that stores anything in >600 are of age >35.

storage Qty > 600, storage.Dept = employee.Dept →

employee.Age > 35
storage.Qty > 600, storage.Dept = employee.Dept, employee.Age ≤ 35 →.

Aggregate constraints:

AC₁: There are 140 employees making more than 45K.
\[ \sigma(\text{employee.Sal} > 45\text{K}) \mid \text{count(\text{employee.Ssn})} = 140. \]

AC₂: Minimum age of a manager is 45 years.
\[ \sigma(\text{employee.Job} = \text{'Manager'}) \mid \text{min(\text{employee.Age})} = 45. \]

AC₃: Cumulative experience of Sales managers is 50 years.
\[ \sigma(\text{employee.Job} = \text{'Manager'}, \text{employee.Dept} = \text{'Sales'}) \mid \text{sum(\text{employee.Experience})} = 50. \]

AC₄: Average age of a manager is 50 years.
\[ \sigma(\text{employee.Job} = \text{'Manager'}) \mid \text{avg(\text{employee.Age})} = 50. \]
Chapter 4

CONSTRAINT UTILIZATION

4.1 Heuristics and inference rules

Before formally describing the query transformation process, we present a brief overview of various heuristic and inference rules used in semantic optimization. The illustrations are based on the example database presented in chapter 3.

We use four heuristic rules as suggested in [King81], namely, restriction elimination, index introduction, scan reduction, and join elimination. The heuristic strategy of join introduction as in [King81] is not used in our approach. In the following illustration we use a quel-like language [Ston76] for expressing queries.

4.2 Restriction elimination:
Remove a restriction from the query, if found redundant.

Query $Q_1$: List all the departments that store benzene in qty more than 400.

Quel Form: retrieve (storage.Dept) where

storage.Material = "Benzene" and

- 22 -
storage Qty > 400.

Rule(s) : storage.Material = "Benzene" \(\rightarrow\) storage Qty > 500.

Query \(Q_1'\) : retrieve (storage.Dept) where

storage.Material = "Benzene".

Result : The unnecessary restriction on the attribute "Qty" of the relation "storage" is eliminated.

4.3 Index Introduction:

Introduce a restriction on an indexed attribute, if implied by the query.

Query \(Q_2\) : Find all the employees who make more than 42K.

Quel Form: retrieve (employee.Ssn, employee.Name) where

employee.Salary > 42K.

Rule(s) : employee.Salary > 42K \(\rightarrow\) employee.job = "Manager".

Query \(Q_2'\) : retrieve (employee.Ssn, Name) where

employee.Salary > 42K and

employee.Job = "Manager".

Result : A new constraint is obtained on the indexed attribute "Job" of the relation "employee".
4.4 Scan Reduction:

Reduce the number of inner scans of the join by obtaining additional restrictions prior to the cross referencing operation.

Query Q₃: List all employees working in departments storing anything in Qty > 625.

Quel Form: retrieve (employee.Ssn, employee.Name) where

   employee.Dept = storage.Dept and
   storage.Qty > 625.

Rule(s): storage.Qty > 600, storage.Dept = employee.Dept → employee.Age>35.

Query Q₃': retrieve (employee.Ssn, employee.Name) where

   employee.Dept = storage.Dept and
   storage.Qty > 625 and
   employee.Age > 35.

Result: The new constraint on attribute "Age" of relation "employee" can be applied to the relation prior to the cross matching step of its join to the relation "storage", thus reducing the qualifying tuples from the relation "employee" and hence the number of scans of the relation "storage".
4.5 Join Elimination:

Eliminate a relation if it is joined to just another relation and none of its attributes contribute to the output.

Query $Q_4$: Get all the materials stored in "d1", in qty $>$ 400, of risk $>$ 2.

Qual Form: retrieve (storage.Material) where

storage.Dept = "d1" and
storage Qty > 400 and
storage.Material = material.Material and
material.Risk $>$ 2.

Rule(s): $storage.Dept = "d1", storage.Material = material.Material \rightarrow material.Risk > 3$

material.Material is a superset of storage.Material

Query $Q_4'$: retrieve (storage.Material) where

storage.Dept = "d1" and
storage Qty > 400.

Result: Join with the relation "material" is eliminated.
Chapter 5

SEMANTIC TRANSFORMATION

5.1 Query

A simple query Q, expressed in the context of a database scheme D, is syntactically similar to a unit clause (fact). The difference is that a unit clause asserts that a goal is true, whereas a query asks whether the goal is true. The variables appearing in a query are implicitly existentially quantified. Shared variables are used as a means of constraining a simple query by restricting the range of a variable.

A substitution is a finite set of pairs of the form $X_i = t_i$, where $X_i$ is a variable and $t_i$ is a term, and $X_i \neq X_j$ for every $i \neq j$, and $X_i$ does not occur in $t_j$, for any $i, j$. The result of applying a substitution $\emptyset$ to a term $A$, denoted by $A\emptyset$, is the term obtained by replacing every occurrence of $X$ in $A$ by $t$, for every pair $X=t$ in $\emptyset$. $B$ is an instance of $A$ if there is substitution $\emptyset$ such that $A\emptyset = B$. Answering a query is the process of finding all the facts that are instances of the query. All such instances form the solution of the query.
Conjunctive queries are practically more relevant than the simple ones. A conjunctive query Q specifies a conjunction of goals posed as a query. Shared variables are used to specify equality restrictions as well as equijoins between terms in conjunctive queries. Inequality restrictions and inequality joins are specified by explicit terms. The explicit inequality operators used in our discussion are from \{\neq, >, \geq\}. The operators \(<\) and \(\leq\) are not explicitly considered because \(a < b\) and \(a \leq b\) can be represented by \(b > a\) and \(b \geq a\) respectively. Similarly, \(a = b\) can be represented by the conjunction of \(a > b\) and \(b > a\).

A query Q, thus, can be considered as a conjunction of join specifications of the form "\(r_1.A_1 \text{ op } r_2.A_2\)" and the restriction specifications of the form "\(r_1.A_1 \text{ op } k\)" where \(r_1, r_2\) are relations, \(A_1, A_2\) are attributes, \(k\) is a constant, and op is one of the comparison operators \{\neq, \geq, >\}. The answer of a query Q is the set of all tuples of the relations referenced in Q that satisfy Q, projected on the specified target attributes of Q.

5.2 Basic query representation

A query Q is represented by a query graph \(G_Q\) which is a directed graph whose vertices are the attributes of the relations (attribute vertices) as well as the constants (constant vertices)
involved in Q. The edges of \( G_q \) are the join and restriction specifications in Q. A join specification "\( r_1.A_1 \) op \( r_2.A_2 \)" is represented by an edge from \( r_1.A_1 \) to \( r_2.A_2 \) with a label op. Similarly, a restriction specification "\( r_1.A_1 \) op k" is represented by an edge from \( r_1.A_1 \) to the constant k with a label op. The direction of an edge identifies the left and right operands of the label associated with it. The edges representing a join specification are referred to as "join edges" whereas the ones denoting the restrictions are called "restriction (constant) edges".

Consider the query Q4 of chapter 4 for the illustration below:

**Query Q4:**

Get all the materials stored in "d1", in qty>400, of risk>2.

**Quel Form:**

```
retrieve (storage.Material) where
    storage.Dept = "d1" and
    storage.Qty > 400 and
    storage.Material = material.Material and
    material.Risk > 2.
```
Graph $G_q$.

The graph $G_q$ is as shown below. Indexed attributes are underlined, target attributes are identified by "?". For the sake of clarity, equalities are represented by undirected single edges rather than pairs of "\$\triangleright\$" edges of opposite directions.

![Graph $G_q$]

**Fig 5.1: Query graph, $G_q$**

5.3 Transitive reduction:

A given query can have syntactically different forms, leading to different query graphs, which are semantically equivalent. Hence it becomes necessary to develop a canonical form of representation, $G_c$, before processing the query. We arrive at
such a canonical representation of the graph by obtaining its transitive reduction [YuOz84], as described in the following discussion.

The edges of the graph \( G_q \) can be classified into three types based on the relational operators represented by them: \( > \)-edges, \( \geq \)-edges, and \( \neq \)-edges. It is possible that there may be multiple parallel edges of more than one type from \( u \) to \( v \) in \( G_q \), where \( u, v \) are distinct vertices. In such cases, the multiple edges from \( u \) to \( v \) can be replaced by a single \( > \)-edge from \( u \) to \( v \), as shown in the following lemma.

**Lemma 5.1:** If there are edges of more than one type from \( u \) to \( v \) in \( G_q \), they can be replaced by a single \( > \)-edge from \( u \) to \( v \).

**Proof:**

An edge \( (u \text{ op } v) \) in \( G_q \), where \( \text{op} \) is one from \( \{ >, \geq, \neq \} \), implies that the clause \( u \text{ op } v \) must be satisfied by \( Q \). The lemma follows from the following implications:

\[
\begin{align*}
\text{u} & > \text{v} \land \text{u} \geq \text{v} \rightarrow \text{u} > \text{v} \\
\text{u} & > \text{v} \land \text{u} \nleq \text{v} \rightarrow \text{u} > \text{v} \\
\text{u} & \nleq \text{v} \land \text{u} \neq \text{v} \rightarrow \text{u} > \text{v} \\
\text{u} & > \text{v} \land \text{u} \neq \text{v} \rightarrow \text{u} > \text{v} \end{align*}
\]
So, without any loss of generality, for any two distinct vertices \( u, v \) in \( G_q \), there exists at most one edge from \( u \) to \( v \).

A path in \( G_q \), from \( u \) to \( v \), is a sequence of distinct edges, \( a_1, \ldots, a_p \), \( p \geq 1 \), such that there exists a corresponding sequence of vertices, \( v_0, v_1, \ldots, v_p \), (\( u \) is \( v_0 \) and \( v \) is \( v_p \)), satisfying \( a_{k+1} = (v_k, v_{k+1}) \) is in \( G_q \) for \( 0 \leq k < p \). Length of a path is the number of edges in that path. A path from \( u \) to \( v \) containing only \( \geq \)-edges (denoted as a \( \geq \)-path) implies an edge \( u \geq v \), as the the relation "\( \geq \)" is transitive. Similarly, a path from \( u \) to \( v \) containing \( \geq \)-edges and \( > \)-edges (denoted as a \( > \)-path) implies an edge \( u > v \). On the other hand, a path of length \( > 1 \) from \( u \) to \( v \), containing a \( = \)-edge does not imply any edge between \( u \) and \( v \). These results are stated in the lemma below.

**Lemma 5.2**: 
\[
\text{If } u \geq w \text{ and } w > v \rightarrow u > v \\
\text{and } u > w \text{ and } w \
\]
\( u \) \( \text{op} \) \( w \) \& \( w \geq v \), where \( \text{op} \) is from \( \{ >, \geq, = \} \), does not imply any relationship between \( u \) and \( v \). \( \Xi \)

Since \( G_q \) is a conjunction of clauses, and \( >, \geq \) are transitive, a \( > \)-path from \( u \) to \( v \) in \( G_q \) implies that the clause "\( u > v \)" must be
satisfied by $Q$. Similarly, a $\geq$-path in $G_q$ implies that "$u \geq v$" is satisfied by $Q$.

A **cycle** in $G_q$ is a path of length at least 2, beginning and ending on the vertex. Note that a self loop, that is, an edge of the form $(u, u)$ is not a cycle. $G_q$ is said to be **acyclic** iff it contains no cycles. A cycle is called a $\geq$-cycle if it contains only $\geq$-edges. Otherwise it is called a $>$-cycle.

A $>$-cycle in $G_q$ is a contradiction, because it implies "$u > v$" and "$v > u$", where $u, v$ are two distinct vertices incident on the cycle. A $\geq$-cycle implies an equality between all the vertices that are incident on it, as it implies "$u \geq v$" and "$v \geq u$" for any pair of incident vertices $u, v$.

Two queries, $Q$ and $Q'$, are said to be **equivalent** if they yield the same answers for all the database instances.

**Transitive closure** $G_q^T$ of $G_q$ is a graph containing all the clauses that are in $Q$ or implied by $Q$. Formally, $G_q^T = (V, E^T)$, where $E^T$ is obtained by successively adding all implied edges to $E$ as in lemma 5.2, and replacing multiple parallel edges between two vertices by a single one, as in lemma 5.1. Two queries $Q$ and $Q'$ are equivalent iff $G_q^T = G_q'^T$. 
Instead of the query graph $G_q$, we use the transitive reduction of the query graph to represent the query $Q$. Intuitively, transitive reduction $G_q^t$ of a query graph $G_q$ is a graph with the fewest number of edges among all the graphs having the same transitive closure as of $G_q$. Hence the query represented by $G_q^t$ is equivalent to the given query $Q$, and $G_q^t$ is a minimal representation of the given query, $Q$.

Construction of transitive reduction is as follows:

1) Group vertices of $G_q$ into equivalence classes. Any two vertices belong to the same equivalence class if there is an equality-implying cycle incident on both the vertices. A vertex forms an equivalence class by itself if there is no cycle incident on it. Transform each equivalence class into a single "super vertex".

2) Edge from a super vertex $V_{S_i}$ to a super vertex $V_{S_j}$ is defined as follows. For any two distinct super vertices (equivalence classes) $V_{S_i}, V_{S_j}$, let $E_{i,j}$ be the set of edges in $G_q$ from vertices in $V_{S_i}$ to vertices in $V_{S_j}$. If $E_{i,j} = \emptyset$, then there is no edge from $V_{S_i}$ to $V_{S_j}$. If all the edges in $E_{i,j}$ are of a single type, then there is an edge of the same type from $V_{S_i}$ to $V_{S_j}$. If the edges in $E_{i,j}$ are of more than one type, then there is a single >-
edge from \( V_{s_i} \) to \( V_{s_j} \), as shown in lemma 5.1. This graph is acyclic, as any cycle in \( G_q \) must within an equivalence class, which in turn is a single super vertex. Let this graph be denoted as AC.

3) Obtain the transitive closure of AC by adding all the implied edges to it, as in lemma 5.2. Replace any parallel edges between super vertices using the results in lemma 5.1.

4) The edges in the above transitive closure are successively examined in any order, and those implied by transitivity are removed. The resulting graph is called the transitive reduction, \( G_q^t \) of \( G_q \). We use the transitive reduction as a canonical condensed representation \( G_c \) of the user specified query graph \( G_q \).

5) Finally, each super vertex is expanded by replacing it by a spanning tree of the vertices in the equivalence class corresponding to that super vertex. Any vertex of the spanning tree may be chosen to connect the inter-supervertex edges. \( \Xi \)

The correctness of this construction is stated in the following lemma, which is given without proof.
Lemma 5.3: The transitive reduction constructed by the above procedure satisfies:

a) \((G_q^t)_T = G_q^T\), and

b) If \(G_{q'}^T = G_q^T\), then for any edge \((u,v)\) in \(G_q^t\), there is an edge \((u,v)\) in \(G_{q'}\) not necessarily of the same type. \(\Xi\)

The canonical condensed representation \(G_c\) for the query graph \(G_q\) illustrated above is shown below. The result of this reduction is that the vertices \((\text{storage.material}, \text{material.material})\) and \((\text{storage.dept}, d_1)\) that are connected with equality edges in \(G_q\) form single multimember nodes in \(G_c\).

![Diagram of canonical condensed graph, G_c](image)

**Fig 5.2: Canonical condensed graph, G_c**
5.4 Sequential phases of semantic transformation

Semantic query transformation is the process of obtaining alternative query forms that are semantically equivalent to the original one. The motivation of semantic optimization is to arrive at a more profitable query yielding the same answer, which could be syntactically different from the original query.

In our approach semantic optimization of a query consists of two major phases, namely, semantic expansion, and semantic reduction. Semantic reduction is composed of two stages, namely, relation elimination and edge elimination. These as well as other auxiliary steps are described below.

5.5 Derivation of canonical condensed form

This first step, as described above, obtains a canonical condensed representation $G_C$ of the query through transitive reduction of join and restriction edges present in $G_Q$. The transitive reduction property of the graph is then retained by the query transformation algorithm in all its following stages by removing syntactically redundant edges and/or merging the equivalence classes. This first stage is independent of any semantic details and depends only on the query and the operator
syntax. Besides arriving at a canonical form of the query, this stage facilitates any early detection of contradictions in join or restriction specifications that could lead to a null answer.

5.6 Semantic Expansion

Semantic expansion iteratively adds any new restriction or join edges implied by the combination of (condensed) query graph and semantic implication constraints. This is achieved by identifying the implication constraints whose antecedent atom(s) are satisfied by the graph and adding the restriction or join edges corresponding to their consequent atom to the query. Each time, the transitive reduction property of the graph is restored if the added edge happens to violate it. From the original form, addition of each such edge takes the query graph through various semantically equivalent forms till it reaches a stage $G_m$ where no more new restrictions or joins could be implied.

The purpose of semantic expansion is to incorporate any useful restrictions (possibly on indexed attributes) or joins that are not present in the original query. This assures that the query contains semantically maximal (and syntactically minimal) set of edges that satisfy both the query and implication constraints.
Semantic expansion of the query graph is illustrated below. Both the antecedent atoms of the second implication constraint (i.e., "storage.Dept = d1" and "storage.Material = material.Material") are satisfied by the query graph, thus making it possible to add the corresponding consequent atom (i.e., "material.Risk > 3") to the graph. Due to the transitive reduction property of the graph, the added edge "material.Risk > 3" supersedes and hence eliminates the existing one "material.Risk > 2".

Fig 5.3: Semantic Expansion, $G_m$. 
5.7 Relation Elimination

Relation elimination stage identifies all semantically redundant relations from $G_m$. Relations identified to be redundant, if any, are removed from the query graph. A relation is considered to be redundant if it becomes dangling so that none of its attributes or restrictions contribute to the answer. Since the query graph is connected, elimination of a relation leads to the removal of its join to the rest of the query graph. A relation elimination is hence considered to be profitable because it eliminates the need of performing a join. The graph, when all the redundant relations are removed from $G_m$, is denoted by $G_{r1}$.

There are various conditions that a relation should satisfy to be classified as redundant. First of all, it should be free from any target attributes of the query because a relation containing target attributes cannot be removed from a query. Second condition is that all the restrictions on non-join attributes should be redundant. By this, all such restrictions could be removed without altering the query semantics. In this stage, the relation will have restrictions, if any, only on join attributes. Third condition is that the relation should have at most one join vertex and the fourth one is that the relation does not have any non-equijoins. Third and fourth conditions allow the
transfer of all the restrictions on the only join attribute to the other side of the respective joins. This makes the relation free from all restrictions. Finally, there should be at least one other relation with a join attribute, say "S.B", in the equivalence class containing the join attribute of this relation, say "R.A", such that the subset constraint "R.A ⊇ S.B" holds.

More formally, a relation R is redundant if satisfies all of the following conditions:

a) R is target-free.
b) All the restriction edges on non-join vertices of R are redundant.
c) R has at most one join attribute.
d) R does not have any non-equi joins.
e) There is at least one other relation with a join attribute, say "S.B", in the join class containing the join attribute of R, say "R.A", such that the subset constraint "R.A ⊇ S.B" holds.

Relation elimination of the graph G_m illustrated above is as shown below. The relation "material" gets qualified as redundant due to the conditions described above. It is free from any target attributes (condition a), and the restriction "material.Risk > 3" is semantically redundant (condition b). The only join attribute of the relation is "material.Material" (condition c) and it is an
equijoin (condition d). The subset constraint "material.Material is a superset of storage.Material" (condition e) completes the requirements for making the relation "material" semantically redundant and hence removable from the query.

![Diagram of relation elimination]

**Fig 5.4: Relation Elimination, G**

### 5.8 Edge elimination

A restriction or join edge is redundant if it is satisfied by the consequent atom of an implication constraint of which all the antecedent atoms are satisfied by the query. Semantically redundant join edges can always be removed from the graph, since their basic purpose is to aid semantic expansion by providing additional paths for information flow. The strategy of removing a redundant restriction edge from a relation largely depends on whether selections will be performed in that relation before joins. This, in turn, depends on whether the join attribute in that relation is indexed or not. If the relation has at least one
indexed join attribute, it is assumed that the restrictions in that relation will be performed along with the join, and not before it. This is because, with a given set of matching values for the join attribute, location of tuples becomes easy through the (indexed) join attribute and the restriction(s) could be checked during the same time. On the other hand, if no join attribute of the relation is indexed, we assume that the selections in that relation will be performed before joins.

In the cases where selections are performed before joins, locally redundant restriction edges on indexed attributes become profitable provided none of the antecedent atoms of the corresponding local implication constraint are on indexed attributes. This is because, the redundant restriction introduces an indexed scan to replace the sequential scan of the relation. Similarly, all the cross redundant restrictions become profitable if selections are performed before joins. The reason is that such a restriction additionally limits the effective size of the relation before the join operation, thus resulting in a scan reduction. A restriction, even though redundant, is considered to be profitable if it is on a join attribute since it may provide a better join strategy.
The graph resulting from deleting all non profitable edges from $G_{r1}$ is denoted as $G_{r2}$. For the query graph $G_{r1}$ shown in the above example, no edge is qualified for elimination. That is, $G_{r2}$ is the same as $G_{r1}$ in this case.

Unlike in the case of relation elimination where all the redundant joins are removed, redundant edges are retained if they are found profitable. But identifying a restriction to be profitable, as mentioned above, depends mainly on the estimation of the sequence of selections and joins. This sequence, in reality, is determined by various factors outside the scope of this work, like relation sizes, optimizer statistics, optimizer intelligence, and sequence of specification of equality joins. This might result in erroneous classification of profitable restrictions at times. But by and large, this strategy provides a simple and reliable method to achieve the identification.

5.9 Conversion from the condensed query graph

When the semantic expansion and reduction are completed, the query graph is converted back from its condensed form to the original one. This is achieved by replacing each multimember node of $G_{r2}$ by any spanning tree on its attribute vertices connected by equijoin edges. Any one attribute vertex, an indexed one if
available, of the multimember node is chosen for joining the spanning tree to other spanning trees or single attribute vertices. Also, the restriction(s) on the multimember node are mapped as restriction(s) on all the attribute vertices. This form of the graph, being the final one, is denoted by $G_f$.

Note that, as in the case of restriction elimination, this strategy of graph conversion could also produce suboptimal results. Cost of equijoins involving three or more attribute vertices depends on the edges in the corresponding spanning tree as well as the order in which they are considered for join. Similarly, selecting an attribute vertex in the spanning tree to join with other vertices could also make a difference in cost. For example, while selecting the edges of the spanning tree, priorities are given to the attributes of these relations which have one or more other joins between them. In general, issues like relation sizes and selectivities should also be considered in selecting the spanning tree for multimember equivalence classes.

The converted form $G_f$ of the graph $G_{r1}$ is as illustrated below. The result is the replacement of the multimember node $(\text{storage.dept}, d_1)$ by its spanning tree. This final graph can be translated to a quel statement:
retrieve (storage.material) where
storage.dept = d₁ and
storage.qty > 400

The end result of the query transformation process, in this example, is elimination of the relation "material" from the original user-specified query.

![Diagram]

Fig 5.5: Converted form, G₁
Chapter 6

QUERY TRANSFORMATION ALGORITHM

In this chapter we formalize the algorithm for semantic transformation, and discuss its correctness and cost saving.

6.1 Algorithm for query transformation

Stage 1: Obtain canonical condensed form - Construct $G_c$ from $G_q$:

The construction, same as the one described in section 5.3, is repeated below:

a) Group vertices of $G_q$ into equivalence classes. Any two vertices belong to the same equivalence class if there is an equality-implying cycle incident on both the vertices. A vertex forms an equivalence class by itself if there is no cycle incident on it. Transform each equivalence class into a single "super vertex".

b) Edge from a super vertex $Vs_i$ to a super vertex $Vs_j$ is defined as follows. For any two distinct super vertices (equivalence classes) $Vs_i, Vs_j$, let $E_{i,j}$ be the set of edges in $G_q$ from vertices in $Vs_i$ to vertices in $Vs_j$. If $E_{i,j} = \emptyset$, then there
is no edge from $V_s_i$ to $V_s_j$. If all the edges in $E_{i,j}$ are of a single type, then there is an edge of the same type from $V_s_i$ to $V_s_j$. If the edges in $E_{i,j}$ are of more than one type, then there is a single $>\!-$edge from $V_s_i$ to $V_s_j$, as shown in lemma 5.1. This graph is acyclic, as any cycle in $G_q$ must within an equivalence class, which in turn is a single super vertex. Let this graph be denoted as $AC$.

c) Obtain the transitive closure of $AC$ by adding all the implied edges to it, as in lemma 5.2. Replace any parallel edges between super vertices using the results in lemma 5.1.

d) The edges in the above transitive closure are successively examined in any order, and those implied by transitivity are removed. The resulting graph is called the transitive reduction, $G^t_q$ of $G_q$. We use the transitive reduction as a canonical condensed representation $G_c$ of the user specified query graph $G_q$.

Stage 2: Obtain semantic expansion, $G_m$:

a) Mark any unmarked implication constraints in which all predicates but one are currently implied by $G_c$. (We denote
this predicate as unimplied predicate). Terminate the stage if no new implication constraint get qualified for marking.

b) If there is at least one marked but unused implication constraint, add an edge corresponding to its unimplied predicate to $G_C$.

c) If any edges are added in step b, restore $G_C$ to its transitive reduction. This may include removal of syntactically redundant edges and/or merger of vertices. Repeat step a.

Stage 3: Eliminate redundant relations to obtain $G_{r1}$:

While there is a relation $R$ that satisfies all of the following:
  a) $R$ is target-free.
  b) All the restriction edges on nonjoin vertices of $R$ are redundant.
  c) $R$ has at most one join attribute.
  d) $R$ does not have any non-equi joins.
  e) There is at least one other relation with a join attribute, say "S.B", in the join class containing the join attribute of $R$, say "R.A", such that the subset dependency "R.A $\supseteq$ S.B" holds.
eliminate \( R \) from the query graph.

**Stage 4: Eliminate redundant edges, to obtain \( G_{r2i} \)**

Remove all the redundant (those implied by the rest of the query graph) join edges.

Remove all the redundant (those implied by the rest of the query graph) restrictions if they are not profitable:

a) A redundant restriction on a join attribute is profitable.

b) If no join attribute of the relation is indexed, then all the cross redundant restrictions in that relation are profitable.

c) If no join attribute of the relation is indexed, then all locally redundant restrictions in that relation are profitable only if they are on indexed attributes and the corresponding antecedent restrictions of the implication constraints are on non indexed attributes.

**Stage 5: Expand multimember nodes of \( G_{r2} \) to obtain \( G_{e} \):**

a) Replace each multimember node of \( G_c \) by a spanning tree on its attribute vertices connected by equijoin edges. While selecting vertices of the spanning tree, assign priorities
for attributes of those relations which have one or more other joins between them.

b) Select any attribute vertex, an indexed one if available, of the multimember node for joining the spanning tree to other spanning trees or single attribute vertices.

c) Map the restriction(s) of the multimember node as the restriction(s) on all the attribute vertices.

6.2 Correctness of the algorithm

Let the original form of the query be denoted by $Q_o$, the expanded form on completion of the expansion stage of the algorithm by $Q_m$, and the final transformed form by $Q_f$.

**Theorem 6.1:** The query forms $Q_o$, $Q_m$, and $Q_f$ are semantically equivalent.

**Proof, $Q_o \iff Q_m$:**

The semantic transformation from $Q_o$ to $Q_m$ takes place in the expansion stage due to the addition of edges to $G_C$ from the consequent atoms of the implication constraints. (Note that
initial conversion from $G_q$ to $G_c$ does not alter the query semantics). Addition of each such edge to $G_c$ can be assumed to transform the graph to a new query form. The transformation from $Q_0$ to $Q_m$ thus constitutes a chain, $Q_0 \rightarrow Q_{o1} \rightarrow Q_{o2} \rightarrow \ldots \rightarrow Q_m$.

Here, the difference between two consecutive forms $Q_{oi}$ and $Q_{oi+1}$ is at most one edge, say $e_i$ (apart from any differences resulted by restoring the graph to its transitive reduction, which does not alter any semantics of the query). Introduction of $e_i$ to $Q_{oi}$ is due to the presence of a set of edges $E_i$ in $Q_i$ such that there exists an implication constraint $E_i \rightarrow e_i$ in the structure $C_i$. So addition of $e_i$ to $Q_{oi}$ does not alter the semantics of $Q_{oi}$. In other words, $Q_{oi}$ and $Q_{oi+1}$ are semantically equivalent. Extending the argument for the entire chain of transformations, it can be seen that $Q_0$ and $Q_m$ are semantically equivalent.

**Proof**, $Q_m \Leftrightarrow Q_f$:

The transformation from $Q_m$ to $Q_f$ is accomplished in the elimination stage of the algorithm. As above, let us assume that the transformation from $Q_m$ to $Q_f$ can be represented by a chain, say $Q_m \rightarrow Q_{m1} \rightarrow Q_{m2} \rightarrow \ldots \rightarrow Q_f$. 
The transformation from $Q_{mi}$ to $Q_{mi+1}$ can be due to elimination of a relation (join) or removal of an edge by the semantic reduction stage of the algorithm.

All the relations and edges qualified for elimination are the ones found semantically redundant, and hence their removal does not alter the semantics of the query. Hence we conclude that $Q_{mi}$ and $Q_{mi+1}$ are semantically equivalent, implying the semantic equivalence of $Q_m$ and $Q_f$.

6.3 Cost comparison of query forms

**Theorem 6.2:** $\text{Cost}(Q_f) < \text{Cost}(Q_o)$ provided the estimation of selection-join sequence is valid.

**Proof:**

If $Q_f$ is different from $Q_o$, let this difference be represented by three components: 1) Set of edges $E_{f+}$ that are present in $Q_f$ but not in $Q_o$, 2) Set of edges $E_{o+}$ that are present in $Q_o$ but not in $Q_f$, 3) Set of relations $R_{o+}$ that are present in $Q_o$ but not in $Q_f$. 
The edges in $E_{f+}$ are syntactically or semantically redundant since they have been added by the algorithm to the initial graph during initial conversion (to $G_o$) or semantic expansion. The fact that they were not eliminated during the semantic reduction implies that they belong to the "profitable" category, provided the estimated selection-join sequence holds good. In this context they represent an additional profit for $Q_f$ as compared to $Q_o$.

All the edges in $E_{o+}$ are also syntactically or semantically redundant because otherwise they would have been retained in $Q_f$ too. The reason for their removal by the semantic reduction stage was that they were not found to be profitable. In other words, the edges in $E_{o+}$ represent an elimination of the non profitable part from the original query, if the estimations on selection-join sequence holds good.

In short, as compared to $Q_o$, $E_{o+}$ represents the edges lost whereas $E_{f+}$ represents edges gained by $Q_f$. The strategy of adding and eliminating the edges always concentrates on adding profitable edges and removing non profitable ones. Both these components thus represent profit provided the sequence estimation of selections and joins are valid.
The set $R_{0+}$ represents a clear profit for $Q_f$ because $Q_f$ does not have the corresponding joins.

To conclude, the cost advantage of $Q_f$ over $Q_o$ can be represented by $C = a_1|E_{f+}| + a_2|E_{0+}| + a_3|R_{0+}|$, where $a_1$, $a_2$, $a_3$ are scaling factors to reflect the relative importance of components as well as validity of the assumption on selection-join sequence. If these assumptions are valid, $C$ represents a positive quantity, and in that case the larger the sets $E_{f+}$, $E_{0+}$, $R_{0+}$ are, the higher is the resulting cost advantage.
Chapter 7

IMPLEMENTATION AND RESULTS

7.1 Implementation details

The optimization algorithm has been implemented on a Vax 8530 running VMS (Version 4). Ingres (Version 5) database management system from Relational Technology Incorporated (RTI) was used for data storage and user interface. Various Ingres utilities (ABF, OSL, Vifred) were used for interface design. The core implementation language was C.

7.2 Architecture

The optimizer consists of three main modules - specification module, maintenance module, and processing module.

The specification module is an interface for end users to specify queries in an interactive mode and get the semantically optimized query forms back. For an ordinary user, this is the only interface to the optimization system. The frame associated with this module has three major sections, one for specifying the initial query form, the second one for displaying the optimized form, and the third one for displaying the run time statistics. A
query consists of a combination of two sets, namely, a set of target attributes and a set of qualifications. Both these sets are entered in individual tabular fields capable of scrolling independently, thus allowing the system to handle any number of target attributes and qualifications. This module also supports an exhaustive error management scheme. Target attributes and qualification specifications entered by the user are validated against the schema details, and any error is reported before passing the information to the processing module. The run time statistics displayed by this module include relative time spent by the processing module on various sections of the optimization algorithm. Figure 7.1 illustrates the frame associated with the specification module. This module is implemented using the Ingres utilities Vifred, ABF, and OSL.
The maintenance module is a background module which is generally invisible to the normal user. A user with maintenance privilege can use this module to manage the details of schema, index information, and semantic rules. The data dictionary containing relation names, their attributes, and index information as well as the semantic details containing implication constraints and subset constraints are stored in tabular data structures. Tabular fields with independent scrolling features are used to display them on the frame. This module also has comprehensive error checking mechanisms to make sure that all the semantics specified by the user are valid for the existing schema definition. The frame associated with the maintenance module is illustrated in Figure 7.2. This module also is implemented using the Ingres utilities Wifred, ABF, and OSL.

<table>
<thead>
<tr>
<th>RELATION NAME</th>
<th>ATTRIBUTE</th>
<th>TYPE</th>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>First</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Empnum</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>Storage</td>
<td>Dept</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Grade</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Salary</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Bonus</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>Dept</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

Fig 7.2 Maintenance Module
The processing module implements the optimization part of the algorithm. The query entered by the user in the specification module is passed to this module after some syntactic analysis and preliminary error checking. The query is then analyzed by the processing module in the context of existing schema and semantic details (relations and rules). The processing module then transforms the query through various sequential stages of the algorithm, namely, transitive reduction, compressed graph formation, semantic expansion, relation elimination, restriction elimination, and spanning tree generation. Errors and contradictions detected at any stage is reported to the specification module, and processing is aborted in such cases. As the processing proceeds, the module also collects various run time statistics. The final form of the query as well as the collected statistics is passed to the specification module upon completion of the processing. The major portion of the processing module is implemented in C language. A small portion is written in Equel (query language supported by Ingres) to communicate with the storage tables where the schema and semantics details are stored.

Currently, all the inter-module communication uses stored tables as the main data structures. Access to the stored information is minimized in the processing module due to efficiency considerations. On the other hand, the other two
modules assign importance to human factors and user friendliness rather than operating speed. Management of errors is uniform and exhaustive in all the three modules, and the errors handled range from simple specification mistakes to complex semantic contradictions.

7.3 Test results

As a basic test of the implementation of semantic query optimizer, we selected the database scheme and the semantic rules introduced in chapter 3. The database contained three relations, as repeated below:

**Schema:** (Indexed attributes are underlined).

employee (San, Name, Dept, Job, Grade, Sal, Bonus, Age)

storage (Dept, Material, Qty)

material (Material, Risk, Storage Limit)

7.3.1 Queries used in the experiment:

We used the queries discussed in chapter 4 as test queries, which are also given below, where $Q_i$ are the original queries and $Q'_i$ the optimized ones.
restriction elimination:

Q1: retrieve (storage.dept) where
   storage.material = "BENZENE" and
   storage.qty > 400

Q1': retrieve (storage.dept) where
   storage.material = "BENZENE"

index introduction:

Q2: retrieve (employee.lname, employee.fname) where
   employee.salary > 42K

Q2': retrieve (employee.lname, employee.fname) where
   employee.salary > 42K and
   employee.job = "MANAGER"

scan reduction:

Q3: retrieve (employee.lname, employee.fname) where
   employee.dept = storage.dept and
   storage.qty > 625

Q3': retrieve (employee.lname, employee.fname) where
   employee.dept = storage.dept and
   storage.qty > 625 and
employee.age > 35

**relation elimination:**

Q4: retrieve (storage.material) where
    storage.dept = "dl" and
    storage.qty > 400 and
    storage.material = material.material and
    material.risk > 2

Q4': retrieve (storage.material) where
    storage.dept = "dl" and
    storage.qty > 400

7.3.2 Test procedure:

The employee table was populated by actual data from a payroll file, and then modified to ensure anonymity. The storage table was filled by a random number generator. The material table was loaded from a chemical data file. All the tables were then extensively modified manually to be consistent with the semantic rules. Following is a brief outline of the profile of the data in the tables:

**employee:**

total number of tuples: 37043
number of tuples where "Salary>42K": 3021
number of tuples where "Job="Manager": 3773
number of tuples where "Age>35" : 24270

storage:
    total number of tuples: 1601
    number of tuples where "material="Benzene": 2
    number of tuples where "Qty>400": 704
    number of tuples where "Qty>625": 18
    number of tuples where "Dept="d1": 278

Material:
    total number of tuples: 1801
    number of tuples where "Risk>2": 898

The four original queries \((Q_1, \ldots, Q_4)\) and their optimized counterparts \((Q_1', \ldots, Q_4')\) represent each of the four transformation heuristics. Several tests were performed to study the optimization costs and execution costs of the above query pairs. Both these costs were measured in terms of two resources consumed by the Ingres process, \_cpu\_ms (cpu time in milli seconds) and \_dio\_cnt (direct i/o requests). Since these parameters are highly influenced by hardware configuration, we do not stress any units.
For us, the matter of relevance is only the relative magnitude of these parameters.

All the four queries \((Q_1,..Q_4)\) were fed to the semantic optimizer to generate the optimized forms \((Q_1',..Q_4')\). This process was repeated several times for each of the queries to measure the optimization cost. The optimization cost (in terms of \_cpu_ms and \_dio_cnt) did not seem to vary much from query to query and we obtained an average of about 180 \_cpu_ms and 30 \_dio_cnt.

Then the four original queries and the corresponding four semantically optimized ones were run against the database tables repeatedly, about 15 times. During each execution, cost of the query was monitored using the above two parameters. Below is a consolidation of the average values from these 15 tests.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_cpu _dio</td>
<td>_cpu _dio</td>
<td>_cpu _dio</td>
<td>_cpu _dio</td>
<td>_cpu _dio</td>
</tr>
<tr>
<td>Q1</td>
<td>657  60</td>
<td>582  50</td>
<td>11.46 17.12</td>
<td>762  80</td>
<td>-15.9 -33.33</td>
</tr>
<tr>
<td>Q2</td>
<td>14461 2571</td>
<td>4227 368</td>
<td>70.77 85.69</td>
<td>4407 398</td>
<td>69.52 84.52</td>
</tr>
<tr>
<td>Q3</td>
<td>138295 4364</td>
<td>91209 3325</td>
<td>34.05 23.80</td>
<td>91389 3355</td>
<td>33.92 23.12</td>
</tr>
<tr>
<td>Q4</td>
<td>15167 88</td>
<td>626 49</td>
<td>95.87 44.25</td>
<td>806 79</td>
<td>94.69 10.22</td>
</tr>
</tbody>
</table>

Fig 7.3: Test results of semantic optimization

7.3.3 Discussion:

It can be observed from the above results that optimized versions of the queries take less execution time (in terms of cpu time and direct i/o requests) as compared to the original ones. The magnitude of saving depends on size of the tables involved as well as the amount of potential optimization possible. This saving is partially offset by the optimization cost which, as reported above, is about 180 _cpu_ms and 30 _dio_cnt. In most circumstances these values are much smaller than the saving in execution cost, thus justifying semantic optimization. On the other hand, if the original query is already in the optimized form, the optimization cost becomes an addition to the execution cost. This is the situation with Q1, where the execution cost of
the optimized query added to the optimization cost is larger than the execution cost of the original query. In such cases, semantic optimization may not be worth the effort.

The savings in the execution cost due to semantic optimization grows with the data size involved in answering the query, whereas the optimization cost remains the same. Hence it is reasonable to assume the optimization cost to be negligible as compared to the savings in the execution cost with very large data sets. Also, if the optimized query is expected to be executed multiple number of times, the optimization cost can be assumed to be amortized over those executions.

If optimization cost becomes comparable with the execution cost, it becomes important to consider a compromise between those two. A detailed analysis of such trade-off between optimization and execution costs is reported in [ShSD88].
Chapter 8

CONSTRAINT MAINTENANCE

Even though the semantic constraints are relatively less frequently updated in comparison with the data itself, an efficient module to manage the semantic modifications is an important part of the optimizer.

8.1 Maintenance of subset constraints

Maintenance of subset constraints is relatively simple. Subset constraints contain exactly two relational attributes and one of the set-comparison operators \( \subseteq, \subseteq^\prime \).

8.1.1 Maintenance Task

Accept a subset constraint if and only if it is not redundant and not contradicting. The constraint is said to be redundant if it can be derived from the existing constraint set. It is said to be contradicting if, when combined with one or more existing constraints produces a null result.

As an example, if \( \rightarrow r.A \subseteq r.B \) and \( \rightarrow r.B \subseteq r.C \) are present in the existing set, a new constraint \( \rightarrow r.A \subseteq r.C \) is
redundant. On the other hand, the constraint \( \rightarrow r.C \subseteq r.A \) is
contradicting.

8.1.2 Assumption

It is assumed that the existing set of subset constraints is
free from redundancy and contradiction. This assumption is
trivially true for a null constraint set.

8.1.3 Data structures

A directed and labelled graph, \( G_s = (V_s, E_s) \), represents the
subset constraints. \( V_s \) is the set of relational attributes
involved in any of the existing subset constraints. A directed
edge \( e \) exists in \( E_s \) from \( v_1 \) to \( v_2 \) (for \( v_1, v_2 \) in \( V_s \)) iff the
subset constraint \( \rightarrow v_2 \subseteq v_1 \) or \( \rightarrow v_2 \subseteq v_2 \) is present in the
existing subset constraint set. The label is used to denote the
operator, \( \subseteq \) or \( \subseteq \). If there are parallel directed edges of the
same label between two vertices, they are replaced by a single
dge with the same direction and the same label. If they are of
different labels, they are replaced by a single edge with the same
direction and "\( \subseteq \)" label. Label "\( \subseteq \)" is said to dominate the label
"\( \subseteq \)". A directed path in the graph is a sequence of directed
dges. A path is said to be of "\( \subseteq \)-type if all the edges in
that path are labelled "c". Otherwise it is called a "c"-type path.

8.1.4 Algorithm for constraint maintenance

Add the vertices \( v_1, v_2 \), corresponding to the new edge (from \( v_1 \) to \( v_2 \)) to the graph, unless they are already present.

If the new edge forms a directed loop with at least one "c" edge, it is a contradicting one.

If there already exists a directed path from \( v_1 \) to \( v_2 \) of the same or dominating type, then the new edge is redundant.

An edge is accepted to the graph (and the corresponding constraint to the semantic set) iff it is neither contradicting nor redundant.

8.1.5 Complexity of the algorithm

The complexity of the above algorithm depends on the complexity of verifying the reachability of a specific vertex from another one using selected types of edges. This can be achieved by finding transitive closure or transitive reduction [YuOz84],
both of which take time bounded by $O(n^3)$ where $n$ is the number of vertices in the graph, which are the relational attributes involved in the subset constraints.

8.2 Maintenance of implication constraints

From here onwards, we denote an integrity constraint by its conjunctive representation and use the words "constraint" and "conjunction" interchangeably, even though the constraint corresponding to the conjunction "$C$" is "$C\rightarrow$". Also, we just use "$C$" in the place of "$C\rightarrow$" to designate a constraint whenever the meaning is unambiguous. In any case, "$C$" represents a conjunction of negated predicates (a clausal integrity constraint).

Notations:

We use the following set of notations in our discussion.

Negation of a conjunction of predicates $c$ is denoted as $\neg c$.

If a set of predicates $c$ is removed from a conjunction $C$, the resulting conjunction is denoted as $C-c$.

The set of values from the domain of attribute $A$ restricted by a predicate $c$ is denoted as $\sigma (c)$.

The null set is denoted by $\emptyset$, the set intersection operator by $\cap$, and the set union operator by $\cup$. 
8.2.1 Deduction system for new constraints

Using various rules of the first order predicate calculus, we can deduce new (redundant) constraints from the existing constraint set. Below we list a set of axioms and inference rules for a deduction system, adapted from Gentzen's work [Gent34, Mann74], which is sometimes referred to as a natural deduction system for first order predicate calculus. The inference rules are divided into two parts: (1) the axioms and basic rules, and (2) rules for the connectives. Each rule is of the form $A \Rightarrow B$, where $A$ and $B$ are conjunctions of predicates, stating that the conjunction $B$ is an integrity constraint if the conjunction $A$ is an integrity constraint.

The rules listed below are from [Mann74] with a slightly modified notation.

The axioms and basic rules:

1. $(A \Rightarrow) \Rightarrow (A, B \Rightarrow)$ (clause introduction)
2. $(A, B \Rightarrow), (\neg A, B \Rightarrow) \Rightarrow (B \Rightarrow)$ (clause elimination)
Rules for the connectives:

3. \((A \rightarrow) \Rightarrow (A, B \rightarrow)\) & introduction

4. \((A, C \rightarrow), (B, C \rightarrow), (\neg A \rightarrow V \neg B \rightarrow) \Rightarrow (C \rightarrow)\) & elimination

5. \((A \rightarrow), (B \rightarrow) \Rightarrow (A V B \rightarrow)\) \(\neg\) introduction

6. \((A V B) \rightarrow \Rightarrow (A \rightarrow)\) \(\neg\) elimination

7. \((A, \neg B \rightarrow), (A, B \rightarrow) \Rightarrow (\rightarrow \neg A)\)

8. \((\neg A \rightarrow), (A \rightarrow) \Rightarrow (\rightarrow B)\)

It is important to note that the above set of rules represent a complete deduction system for propositional calculus [Mend64, Mann74(p111)]. In other words, every valid implication integrity constraint \(C\), with respect to a given constraint set \(S\), can be deduced using the above inference rules. It is possible to derive other inference rules (eg: transitivity of implication) from the above basic set. A detailed discussion can be found in [Mann74].

In the above list, the connectivity rules (3-8) can be derived from the set of basic rules (1,2), definition of implication operation (ie, \(A \rightarrow B\) can be defined as \(\neg A V B\)), and DeMorgan's laws. The rules of the basic set itself belong to two categories, augmentation and transitivity.

Augmentation (rule 1) refers to the uninteresting process of appending arbitrary predicates to an existing conjunction. Since
the existing conjunction - being an integrity constraint - always evaluates to false, all of its superset conjunctions also evaluate to false, thus technically qualifying to be integrity constraints. In rule 1, "A" represents the existing conjunction and "B" represents a conjunction of arbitrary predicates. As an example, consider an integrity constraint 'employee.Sal > 40 K, employee.Job ≠ "Manager"' which states that everyone who makes more than 40K is a manager. It is trivial to generate a new constraint by combining the predicate 'employee.Age > 40' to the above constraint to get "employee.Sal > 40K, employee.Age > 40, employee.Job ≠ "Manager"", which states that everyone who makes more than 40K and over 40 years is a manager.

The following lemma formalizes this result.

Lemma 8.1: For a set of integrity constraints S, and an integrity constraint C, if $S \rightarrow C$, then $S \rightarrow C^\sim$, for any $C^\sim$ where $C^\sim \geq C$.

Transitivity, (rule 2) the other way of generating new constraints, depends on the transitivity of the implication operation. Two constraints, $C_1$ and $C_2$, can be used to transitively generate a new one if there exists conjunctions $C_1, C_2$, where $C_1$ is a subset of $C_1$ and $C_2$ a subset of $C_2$, such that $C_1$ and $C_2$ are complements of each other. The new constraint $C$ is
defined as the conjunction of all the predicates from C₁ and C₂ excluding the ones in c₁ and c₂. In rule 2, "A,B" and "¬A,B" represent C₁ and C₂ respectively, with "A" representing c₁ and "¬A" representing c₂, to generate a new constraint "B" representing C. For example, consider the constraints IC₁ and IC₂ of the example database:


IC₁ and IC₂ can be used (with c₁ = 'employee.Job ≠ "Manager"' and c₂ = 'employee.Job = "Manager"') to obtain:


The proof that C is an integrity constraint is as follows:
Since C₁ is an integrity constraint, (C₁→c₁) → ¬(c₁).
Since C₂ is an integrity constraint, (c₂) → ¬(c₂→c₂).
Since c₂ = ¬c₁, and by transitivity, (C₁→c₁) → ¬(C₂→c₂).
In other words, "(C₁→c₁),(C₂→c₂)", which is C, is an integrity constraint.
Tautology conjunctions

We define tautology conjunctions as the ones that are satisfiable as integrity constraints by all the database instances, irrespective to the semantic rules. In such tautology constraints all the predicates restrict the same attribute. For simplicity of discussion, we consider tautology conjunctions with only two predicates, and the generalization to more than two predicates is straightforward. As an illustration, the conjunction "employee.Grade>20, employee.Grade<18" is a tautology, as it represents the implication "employee.Grade>20 → employee.Grade<18", or equivalently, "employee.Grade<18 → employee.Grade<20", which is true for any database instance with any set of semantics.

Generating weaker constraints from the existing ones can be technically considered as a special case of transitivity involving tautology conjunctions. The above tautology conjunction can be used with 'employee.Job="Manager", employee.Grade<20' to generate a weaker constraint 'employee.Job="Manager", employee.Grade<18'.

We formalize two important properties of tautology conjunctions in the following lemmas.
Lemma 8.2: For a tautology conjunction \( T \) with predicates \( t_1, t_2 \),
\[ \sigma(-t_1) + \sigma(-t_2) = U, \]
where \( U \) denotes the universe of the domain of
the attribute restricted in \( t_1 \) and \( t_2 \).

Proof:
As \( T \) is a tautology, for all the database states, \( t_1, t_2 \rightarrow \).
In other words, \( \sigma(t_1) \& \sigma(t_2) = \phi \).
By taking complements, we get \( \sigma(-t_1) + \sigma(-t_2) = U \). \( \Xi \)

In the above example:
\[ \sigma(\text{employee.Grade}\leq20) + \sigma(\text{employee.Grade}\geq18) = U. \]

Lemma 8.3: A tautology conjunction \( T \), when transitively combined
with a conjunction \( C \), replaces a predicate \( c \) in \( C \) with \( c' \) such
that \( \sigma(c') \subseteq \sigma(c) \).

Proof:
Let \( T \) be \( t_1, t_2 \rightarrow \). As \( T \) transitively interacts with \( C \), \( C \) has
a predicate \( c \) such that \( c \) is the complement of \( t_1 \) or \( t_2 \). Let \( c \) be
the complement of \( t_1 \). Then, by definition of transitivity, the
transitively generated constraint contains all the predicates from
\( C \), with \( c \) replaced by \( t_2 \). Let \( U \) represent the universe of the
domain of the attribute restricted in \( c \), \( t_1 \), and \( t_2 \). We have:

1. \( \sigma(t_1) + \sigma(t_2) \subseteq U \). (true for any sets \( \sigma(t_1), \sigma(t_2) \))
(2) \( \sigma(t_1) \wedge \sigma(t_2) = \phi \). (as \( T \), i.e., "t_1, t_2\), is a tautology)

(3) \( \sigma(t_1) \vee \sigma(c) = U \). (as \( t_1 \) and \( c \) are complements)

(4) \( \sigma(t_1) \wedge \sigma(c) = \phi \). (as \( t_1 \) and \( c \) are complements)

Hence it follows that \( \sigma(t_2) \equiv \sigma(c) \), or \( \sigma(c') \equiv \sigma(c) \). \( \Xi \)

In the above example, the tautology constraint replaces the predicate "employee.Grade\leq20" by "employee.Grade<18", and \( \sigma(employee.Grade<18) \subseteq \sigma(employee.grade\leq20) \).

It is also easy to observe from the above discussion that a conjunction \( C \) is implied by a set of conjunctions \( S \) iff a subset of \( C \) is implied by \( S \).

8.2.2 Intra-constraint redundancy/contradiction:

The first task in maintaining implication constraints is to make sure that no constraint contains any redundant or contradicting predicates. Constraint maintenance system rejects or notifies the user if the new constraint contains redundant or contradicting predicates.

A predicate \( c \) in a constraint \( C \) is said to be redundant if \( C - c \rightarrow c \). Similarly, the predicate is in contradiction if \( C - c \rightarrow \neg c \).
In the previous sections, a generalized representation of implication constraints was used where the consequent predicate was complemented and moved to the antecedent side. This helped the treatment of the implication constraint as a simple conjunctive set of predicates. This generalization is unambiguous only if the constraint is free from any redundant or contradicting predicates. As an illustration, consider the following two implication constraints:

1) e.\text{Sal}<30K, e.\text{Sal}<40K, e.\text{Job}=\text{'Manager'} \rightarrow e.\text{Grade}>20.

2) e.\text{Sal}<30K, e.\text{Job}=\text{'Manager'}, e.\text{Grade}<20 \rightarrow e.\text{Sal}\geq40K.

When the consequent predicate is complemented and moved to the antecedent side, both these implication constraints correspond to the same conjunctive set of predicates. But, in their original forms, the constraints represent different semantics. The first constraint can be written as "e.\text{Sal} < 30K, e.\text{Job} = \text{'Manager'} \rightarrow e.\text{Grade} > 20", making the predicate "e.\text{Sal} \leq 40K" redundant. The second constraint represents a contradiction, since there can be no tuple in the database with "e.\text{Sal} < 30K" and "e.\text{Sal} \geq 40K".

In other words, if the original constraint contains redundant or contradicting predicates, the generalized representation as a conjunctive set of predicates does not hold good. The underlying reason for this discrepancy can be sketched as follows. The
conjunctive representation of the implication constraint is based on the classical interpretation of implication. Classical implication, in terms of \( F \) and \( T \) (False, True) holds good in three cases, viz, \( \{T \rightarrow T, F \rightarrow F, F \rightarrow T\} \). From a database point of view, the implications are used to derive additional truth from an given set of truths. The truth value corresponding to the consequent part of the constraint is considered to hold good only if the antecedent part is true. This makes the last two cases of classical implication, \( \{F \rightarrow F, F \rightarrow T\} \), irrelevant to our case. An implication constraint containing redundant or contradicting predicates depends on these two cases for its existence. A contradiction results in a "False" in the antecedent part, thus making the implication depend on "\( F \rightarrow F \mid T \)". In the case of redundancy, as the redundant predicate can be complemented and moved to the consequent side, the implication has to depend on "\( F \rightarrow T \)".

This observation conforms with the results related to constructivistic logic reported in [Bry89]. The interpretational differences of implication between classical logic and logic programming is analyzed in the above paper. Logic programming considers axioms like "\( \neg A \rightarrow A \)" to be false, whereas classical logic would interpret it as "\( A \rightarrow A \)". From a database point of view too, constructs like "\( \neg A \rightarrow A \)" are invalid, since there can be no
tuple where "$A$" and "$\neg A" hold. As compared to classical logic, logic programs restrict the concept of proof which, in turn, requires either to restrict the axioms or to rely on non-classical inference principles [Bry89].

Our deviation from classical logic is also in conformation with the intuitionistic logic discussed in [Fitt69], [McC88], [BoM89]. Intuitionistic logic can be viewed as a subset of classical logic. Unlike classical logic, intuitionistic implication is not defined in terms of disjunction and negation.

The new constraint specified by the user is first analyzed for intra-constraint redundancy and contradiction. In those cases, conforming the above discussion, we take a slightly different approach from the classical interpretation of implication and try to remove any redundant predicates. Until the constraint is free from redundancy and contradiction, we do not consider its conjunctive representation.

We consider two distinct cases of redundancy/contradiction - one within the antecedent predicates and the other between the antecedent predicates and the consequent predicate. Let the new constraint specified by the user be represented as $P_1, \ldots, P_n \rightarrow P_C$ or, as an abbreviation, as $P \rightarrow P_C$. In the following discussion
we denote the conjunction of all the predicates in \( P \) other than \( p_i \) by "\( P - p_i \)".

\( a) \) redundancy/contradiction within the antecedent predicates:

First, the antecedent predicates \( p_1, \ldots, p_n \) are analyzed for redundancy or contradiction. The antecedent is represented by a directed graph \( G = (V,E) \). The \textit{vertex set} \( V \) represents the variables, constants, and all the unique combinations of the variables and constants present in the conjunction. Note that we use distinct vertices for different constants belonging to the same domain as well as for the same variable in combination with different constants. The \textit{edge set} \( E \) represents explicit as well as implicit comparisons. Explicit comparisons are the ones present in the conjunction. Implicit comparisons are between constants of the same domain and between the same variables in combination with the constants of the same domain. The edges are labelled by their comparison operator. The operators are restricted to \( >, \geq, \) and \( \neq \). The "\( >\)" and "\( \geq \)" edges are directed, their direction representing the direction of the comparison operator, whereas the "\( \neq \)" are undirected. Each edge of \( E \) is then consecutively examined in any order to see whether it is implied by or contradicting to the rest of the graph. A detection of contradiction flags the constraint to be rejected. Otherwise, all
the implied edges are removed from the graph, and then it is converted back to the predicate form to represent the antecedent free from redundancy or contradiction.

As an illustration, the constraint "e.Sal < 30K, e.Sal ≤ 40K, e.Job = 'Manager' → e.Grade > 20", when processed as above, becomes "e.Sal < 30K, e.Job = 'Manager' → e.Grade > 20", as the predicate "e.sal ≤ 40K" is detected as redundant.

b) redundancy/contradiction between the antecedent predicates and the consequent predicate:

Here we assume that the constraint \( p_1, \ldots, p_n \rightarrow p_c \) (abbreviated as \( P \rightarrow p_c \)), processed as above, has the antecedent free from any redundancy and contradiction. Now we analyze the interaction between the antecedent predicates and the consequent predicate. A redundancy or contradiction can result if the attribute restricted by the consequent predicate \( p_c \) is also restricted by one or more predicates in the transitive closure of \( P \). Let there be \( k \) such predicates in the transitive closure that restrict the same attribute as the consequent one does. Without any loss of generality let these predicates be \( p_1, \ldots, p_k \). For each \( p_i \), \( 1 \leq i \leq k \), we analyze four different cases, viz:

1) \( \sigma(p_i) \) and \( \sigma(p_c) \) are non-overlapping.
2) \( \sigma(p_i) \subseteq \sigma(p_c) \).
3) \( \sigma(p_i) \supset \sigma(p_c) \).
4) \( \sigma(p_i) \) and \( \sigma(p_c) \) are overlapping, but neither is a superset of the other.

To study these cases in their most general forms, we assume the domains of \( p_i \) and \( p_c \) to be ordered, thus allowing us to use the comparison operators (">", "\geq" etc) meaningfully.

**Case 1** There is at least one \( p_i \) such that \( \sigma(p_i) \) and \( \sigma(p_c) \) are non-overlapping.

*Eg: e.Sal<30K, e.Job='Manager', e.Grade<20 \rightarrow e.Sal>40K.*

The constraint is contradicting and is hence rejected, as it can be true only with the interpretation \( F \rightarrow T \).

In general terms, since \( p_i \) and \( p_c \) cannot be true simultaneously, neither can \( P \) and \( p_c \). The existence of \( p_c \) does not follow from that of \( P \). In other words, for \( p_c \) to be true, \( P \) is irrelevant, since they cannot interact. As \( \sigma(P) \) and \( \sigma(p_c) \) are non-overlapping, any implication with \( P \) as an antecedent and \( p_c \) as a consequent (or vice versa) cannot hold. Classical interpretation of implication can still consider it to be valid, as it fits in the category of "\( F \rightarrow T\)\|\( F \)". In our case, the constraint is rejected as contradicting.
Case 2] There is at least one \( p_i \) such that \( \sigma(p_i) \subseteq \sigma(p_c) \)

Eg: e.Sal < 30K, e.Job = 'Manager', e.Grade <= 20 \( \rightarrow \) e.Sal < 40K.

The constraint is redundant and is hence rejected, as it is obviously and unconditionally true that the salary is less than 40K if it is less than 30K.

Formally, the fact that \( \sigma(p_i) \subseteq \sigma(p_c) \) unconditionally implies \( p_i \rightarrow p_c \) which, in turn, implies \( P \rightarrow p_c \). Hence we consider the constraint to be redundant.

Case 3] There is a \( p_i \) such that \( \sigma(p_i) \supseteq \sigma(p_c) \)

Eg: e.Sal < 30K, e.Job = 'Manager', e.Grade <= 20 \( \rightarrow \) e.Sal < 20K.

The predicate \( p_i \) is neither redundant nor contradicting. The redundancy or contradiction of the constraint itself depends on the other \( p_i \)'s of the antecedent part, if any. The example says that there is no manager with grade less than or equal to 20 with a salary between 20K and 30K.

Case 4] There is a \( p_i \) such that \( \sigma(p_i) \) and \( \sigma(p_c) \) are overlapping, but neither is a superset of the other.

Eg: e.Sal < 30K, e.Job = 'Manager', e.Grade <= 20 \( \rightarrow \) e.Sal > 20K.

The predicate \( p_i \) is redundant in the constraint. Consider the set of tuples satisfying "e.Job = 'Manager', e.Grade <= 20"
(ie, without antecedent predicate $p_i$). Partition this set of tuples into two as: 1) tuples with "e.Sal < 30K" and 2) tuples with "e.Sal ≥ 30K". For the first partition, the consequent "e.sal > 20K" holds, as per the constraint. For the second partition, the consequent "e.Sal > 20K" holds, as per the discussion in case 2 above. In other words, the consequent holds for all the tuples satisfying "e.Job = 'Manager', e.Grade ≤ 20", thus making the antecedent predicate "e.Sal < 30K" redundant in the constraint. As in case 3, the redundancy or contradiction of the constraint itself depends on any other $p_i$'s the antecedent part may have. If $p_i$ is a user-supplied predicate, then it can be removed from the constraint. On the other hand, if it is an implied by the user-supplied set, then the user is informed about the situation.

The following lemma shows that the predicate $p_i$ is redundant in case 4.

**Lemma 8.4:** If there is a predicate $p_i$ such that $\sigma(p_i)$ and $\sigma(p_c)$ are overlapping, but neither is a superset of the other, then the predicate $p_i$ is redundant.
Proof:

Let us denote the set of tuples satisfying the conjunction of all the antecedent predicates other than $p_i$ (i.e., $P\neg p_i$) by "$U$" (the universal set).

Let the predicates $p_i$ and $p_c$ be expanded over the set $U$ as "$v$ op$_i$ $k_i$" and "$v$ op$_c$ $k_c$" respectively. Here $v$ is the common attribute restricted, op$_i$ and op$_c$ are the comparison operators, and $k_i$ and $k_c$ are constants. We claim that neither of the operators op$_i$ and op$_c$ can be ", because in such a case any overlap between $\sigma(p_i)$ and $\sigma(p_c)$ would make one (with the ", operator) a subset of the other, contradicting our assumption. If the operator is ", the range represented by the predicate extends to infinity on either directions. With any other remaining operators ($>,\geq,<,\leq$), the range represented by the predicate extends to infinity on one direction. With $\sigma(p_i)$ and $\sigma(p_c)$ of overlapping ranges extending to infinity at least on one direction, the condition that none of them is a superset of the other implies that $\sigma(p_i)$ and $\sigma(p_c)$ together cover the entire range of $v$ over $U$.

Hence we have:

1. $\sigma(p_i) + \sigma(p_c) = U$ (from the above discussion).
2. $\sigma(p_i) \cap \sigma(p_c) \neq \emptyset$ (as they are non-overlapping).
(3) \( p_i \rightarrow p_c \) for \( U \) (from the constraint \( P \rightarrow p_c \), and definition of \( U \)).

(4) \( \sigma(\neg p_i) \subseteq \sigma(p_c) \) for \( U \) and \( \sigma(\neg p_c) \subseteq \sigma(p_i) \) for \( U \) (from 1 and 2 above).

(5) \( \neg p_i \rightarrow p_c \) for \( U \) (from 4 above).

(6) \( \rightarrow p_c \) for \( U \). (from 3 and 5 above).

(7) \( p \rightarrow p_i \rightarrow p_c \) (from 6).

In other words, the antecedent predicate \( p_i \) is redundant in the constraint. \( \Xi \)

Each \( p_i \), \( 1 \leq j \leq n \), is tested for the four cases above. If case 1 or case 2 is true for any \( p_i \), the constraint is not accepted. If case 3 is true, the predicate is retained and redundancy/contradiction of the constraint is decided based on any other \( p_i \)'s. If case 4 is true, the predicate is removed from the constraint, and then as in case 3, the redundancy/contradiction is decided by any remaining \( p_i \)'s.

8.2.3 Inter-constraint redundancy/contradiction:

A conjunction of predicates (integrity constraint) \( C \) is said to be redundant with respect to a set of conjunctions of predicates \( S \) if \( C \) is implied by \( S \). In other words, \( C \) is said to
be redundant with respect to \( S \) if the set of databases satisfying \\( S \) and the ones satisfying \( S + \{ C \} \) are the same.

A conjunction of predicates (integrity constraint) \( C \) is said to be in contradiction with respect to a set of conjunctions of predicates \( S \) if \( \neg (C) \) is implied by \( S \). In other words, \( C \) is said to be in contradiction with respect to \( S \), if there is no database that satisfies \( S \) and \( C \).

The task of inter-constraint maintenance is to ensure that the constraint set is always free from redundancy and contradiction. This assumption is trivially true with an empty constraint set. If \( S \) is the existing set of constraints and \( C \) is the new constraint, \( C \) is tested for redundancy (ie, \( S \rightarrow C \)) and contradiction (ie, \( S \rightarrow \neg C \)) before acceptance. If accepted, existing constraints are tested for redundancy with \( C \), and removed from the set if found redundant. As an example, assume that the following two constraints exist in the semantic set:

\[
\text{employee.Sal} > 40K \rightarrow \text{employee.Job} = "Manager" \\
\text{employee.Sal} > 40K \rightarrow \text{employee.Grade} > 20
\]

Consider a new constraint \( \text{employee.Job} = "Manager" \rightarrow \text{employee.Grade} > 20 \), which is neither redundant nor contradicting. However, when added to the set, it makes the constraint \( \text{employee.Sal} > 40K \rightarrow \text{employee.Grade} > 20 \) redundant.
Since the constraint set may contain constraints that do not contribute in making the new constraint redundant (or contradicting), we first identify a sufficient subset of the constraint set for this purpose. This identification restricts the set of constraints to be considered for redundancy/contradiction checking, thus reducing its complexity. As a first step, the existing constraint set is partitioned into equivalence classes.

**Equivalence classes:**

Two constraints $IC_i$, $IC_j$ of the constraint set $S$ are said to be **connected** if any of the following conditions is satisfied:

1. $i = j$, i.e., a constraint is connected to itself.
2. there exist predicates $c_i, c_j$ such that $c_i \subseteq IC_i$, $c_j \subseteq IC_j$, and $\sigma(c_i) \cup \sigma(c_j) = U$, where $U$ represents the universe of the domain of the attribute restricted by $c_i$ and $c_j$.
3. there exists a constraint $IC_k$ such that $IC_i$ is connected to $IC_k$ and $IC_k$ is connected to $IC_j$.

**Connectedness,** as defined above, is an equivalence relation since it is reflexive, symmetric, and transitive. Hence, the set $S = \{IC_1, IC_2, \ldots, IC_n\}$ of existing constraints can be partitioned
into equivalence classes such that two constraints \( IC_i, IC_j \) belong to the same equivalence class iff they are connected.

The constraint set of the example database can be partitioned into four equivalence classes \( E_1, E_2, E_3, E_4 \), as:

\[
\begin{align*}
E_1: & \quad IC_1: \text{employee.Sal } > 40K, \text{ employee.Job } \neq \text{"Manager" } \rightarrow \\
& \quad IC_2: \text{employee.Job } = \text{"Manager"}, \text{ employee.Grade } < 20 \rightarrow \\
E_2: & \quad IC_3: \text{storage.Dept}=d_1, \text{ storage.Material}=\text{material.Material, material.Risk} \leq 3 \rightarrow \\
E_3: & \quad IC_4: \text{storage.Material } = \text{"Benzene"}, \text{ storage.Qty } \leq 500 \rightarrow \\
E_4: & \quad IC_5: \text{storage.Qty} > 600, \text{ storage.Dept}=\text{employee.Dept, employee.Age} \leq 35 \rightarrow
\end{align*}
\]

**Lemma 8.5:** No two constraints from different equivalence classes can transitively interact.

**Proof:**

Suppose there are two constraints, \( C_1, C_2 \), from different equivalence classes that transitively interact, possibly through tautology constraints. We consider two distinct cases. In the first case, let \( C_1 \) and \( C_2 \) transitively interact directly, without any intermediate tautology constraint. By definition of transitivity, there exist two predicates \( c_1, c_2 \), which are complements of each other, such that \( c_1 \sqsubseteq C_1 \) and \( c_2 \sqsubseteq C_2 \). This
implies that \( \sigma(c_1) + \sigma(c_2) = U \), where \( U \) represents the universe of the domain of the attribute restricted in \( c_1 \) and \( c_2 \). In the second case, suppose the interaction between \( C_1 \) and \( C_2 \) involves tautology constraints. In this case, there exists a tautology constraint, say \( T \), such that \( C_1 \) interacts with \( T \), and \( T \) interacts with \( C_2 \). By definition of transitivity, \( c_1 \subseteq C_1 \), \( c_2 \subseteq C_2 \), and \( t_1, t_2 \subseteq T \), such that \( c_1, t_1 \) are complements of each other, and \( c_2, t_2 \) are complements of each other. As \( T \) is a tautology constraint, from lemma 8.2, we have \( \sigma(\neg t_1) + \sigma(\neg t_2) = U \), where \( U \) represents the universe of the domain of the attribute restricted in \( t_1 \) and \( t_2 \). As \( c_1, c_2 \) are complements of \( t_1, t_2 \) respectively, we get \( \sigma(c_1) + \sigma(c_2) = U \).

In both the cases, we have \( \sigma(c_1) + \sigma(c_2) = U \), making the constraints \( C_1 \) and \( C_2 \) connected, by the definition of connectedness. This contradicts the initial assumption that they are in different equivalence classes. Hence we conclude no two constraints from different equivalence classes transitively interact. \( \Xi \)
Minimal subset of C implied by S:

Consider a constraint set $S = \{IC_1, IC_2, ..., IC_n\}$ and a constraint (conjunction) $C$ that is implied by $S$. A subset $C'$ of $C$ is said to be a minimal subset of $C$ implied by $S$ iff $C'$ is implied by $S$ and no proper subset of $C'$ is implied by $S$.

As an illustration, with the constraint set $S$ as in the example consider a new constraint $C$ as:


Constraint $C$ is implied by $S$. But from $IC_1$ and $IC_2$, we can obtain a constraint:

"employee.Sal>40K, employee.Grade<20 →."

The above constraint, when transitively combined with tautology constraints ("employee.Sal≤40K, employee.Sal>45K →" and "employee.Grade≥20, employee.Grade<18 →") generates the following constraint:

"employee.Sal>45K, employee.Grade<18 →."

Let this constraint be denoted as $C'$. As $C' \subseteq C$, and no proper subset of $C'$ is implied by $S$, $C'$ is said to be a minimal subset of $C$ implied by $S$. 
Lemma 8.6: If \( C' \) is a minimal subset of the constraint \( C \) (or \( \neg C \)) implied by the constraint set \( S \), then it is implied by the constraints belonging to a single equivalence class.

Proof:

Since \( C' \) is the minimal subset of \( C \) implied by \( S \), it is not generated by augmentations. So \( C' \) is an existing constraint itself, or is generated by transitivity, possibly involving tautology constraints. If it is an existing constraint, the proof is trivial. Assume it to be a transitively generated one. By lemma 8.5, no two constraints from different equivalence classes can transitively interact. Hence we conclude that \( C' \) is generated by the constraints belonging to a single equivalence class. \( \Sigma \)

So, if \( C' \) is a minimal subset of \( C \) implied by the constraint set \( S \), we can as well say that \( C' \) is a minimal subset of \( C \) implied by the equivalence class \( E \).

For illustration, with the constraint set \( S \), consider the new constraint \( C \) as:

"employee.Sal>45K, employee.Grade<18, employee.Dept='Sales', employee.Age<30 \rightarrow."

The minimal subset \( C' \) of \( C \) implied by \( S \) is:

"employee.Sal<45K, employee.Grade<18\rightarrow."
It can be seen that $C'$ is implied by the equivalence class $E_1$, containing the constraints:

$IC_1$: employee.$Sal>40K$, employee.$Job=\text{"Manager"}$ →

$IC_2$: employee.$Job=\text{"Manager"}$, employee.$Grade<20$ →

From this example, we also make an important observation that every predicate $c$ of $C'$ has a corresponding predicate $i$ in some constraint $I$ of the equivalence class $E$ such that $c \rightarrow i$. Note that:

employee.$Sal>45K$ (of $C'$) → employee.$Sal>40K$ (of $IC_1$).

employee.$Grade<18$ (of $C'$) → employee.$Grade<20$ (of $IC_2$).

Following lemma formalizes this result:

**Lemma 8.7:** For each predicate $c$ in a minimal subset $C'$ of $C$ implied by an equivalence class $E$, there exists a constraint $I$ in $E$ and a predicate $i$ in $I$ such that $c \rightarrow i$.

**Proof:**

As $C'$ is a minimal subset of $C$ implied by $E$, it is not generated by augmentations. So $C'$ is generated by the constraints from $E$ through transitivity. Transitivity must be one of the following types:
1) Transitivity involving the constraints only from E. By definition of transitivity, any new constraints generated must have all their predicates present in some constraint(s) of E.

2) Transitivity involving the constraints from E and tautology constraints. From lemma 8.3, a tautology constraint T, when transitively combined with a constraint I, replaces a predicate i in I with i' such that σ(i') ⊆ σ(i).

So, each predicate c of C' must have a corresponding predicate i in some I of E such that either c = i or σ(c) ⊆ σ(i). In other words, for each c, there exists an i such that c → i. ∎

8.2.4 Relevant subset of a constraint C:

For each equivalence class E_j of S, we define relevant subset C_j of C as follows. For an equivalence class E_j, a predicate c of C belongs to its relevant subset C_j iff there is a predicate i in a constraint I of E_j such that c → i.

Lemma 8.8: S → C iff there exists a j such that E_j → C_j.
Proof:

Assume that there exists at least one $j$ such that $E_j \rightarrow C_j$. As $C \supseteq C_j$, from lemma 8.1, it follows that $E_j \rightarrow C$. As every constraint of $E_j$ is a constraint of $S$ also, we see that $S \rightarrow C$.

Assume that $S \rightarrow C$. Let $C'$ be a minimal subset of $C$ implied by $S$. By lemma 8.6, $C'$ is implied by the constraints belonging to a single equivalence class, say $E_j$. From lemma 8.7, for each predicate $c$ of $C'$, there exists a constraint $I$ in $E_j$ and a predicate $i$ in $I$ such that $c \rightarrow i$. Thus, by definition of relevant subsets, all the predicates of $C'$ should belong to the relevant subset $C_j$. In other words, $C' \subseteq C_j$. As $E_j \rightarrow C'$, we conclude that $E_j \rightarrow C_j$. \[\Xi\]

As an example, consider the new constraint $C$:

"employee.Sal > 45K, employee.Grade < 18, employee.Dept = 'Sales', employee.Age < 30 \rightarrow". The relevant subset $C_1$ for each equivalence class $E_i$ are:

$C_1$: \{employee.Sal > 45K, employee.Grade < 18\}

$C_2$: \{

$C_3$: \{

$C_4$: \{employee.Age < 30\}
Note that the predicate "employee.Dept = 'Sales'" does not belong to any relevant subsets. In this particular example, since \( E_1 \rightarrow C_1 \), \( C \) is redundant.

8.2.5 Constraint derivation from equivalence classes

Once the relevant subset of the new constraint \( C \) for implication checking is identified for each of the equivalence classes, the task is to verify whether any of the equivalence classes actually implies its corresponding relevant subsets.

Let a particular equivalence class contain the constraints \( 'C_1 \rightarrow', ....., 'C_n \rightarrow' \). If the relevant subset \( C' \rightarrow \) is implied by this set, \( \neg(C_1), ....., \neg(C_n) \rightarrow \neg(C') \) must be a tautology or, in other words, \( \neg(C_1), ....., \neg(C_n), C' \) must be unsatisfiable. Since \( C_i 's \) are conjunctions of predicates, \( \neg(C_1), ....., \neg(C_n) \)" is a conjunction of disjunctions and can be rewritten as a disjunction of conjunctions of predicates from \( C_i 's \). Each of the conjunctions will contain \( n \) predicates, and there will be \( t_1 \times ... \times t_n \) such conjunctions in the disjunction, where \( t_i \) is the number of predicates in \( C_i \). When \( C' \) is conjoined with disjunction, the resulting disjunction will have the same number of conjunctions, i.e., \( t_1 \times ... \times t_n \), each with \( n+t \) predicates each, where \( t \) is the number of predicates in \( C' \).
Similarly, testing whether the set of sufficient constraints implies the complement of the new constraint is equivalent to testing the unsatisfiability of "¬C_1,...,¬C_n,¬(C')". This represents a disjunction of conjunctions from C_i's and C. Each conjunction will have n+1 predicates, and there will be t_1...t_n*t such conjunctions in the disjunction.

In both these cases, the number of conjunctions increases exponentially with the number of constraints in the equivalence class, where as the number of predicates in each conjunction has a linear growth.

For illustration, let us consider the testing of E_1 → C_1, where:

E_1: employee.Sal > 40K, employee.Job ≠ "Manager" →
    employee.Job = "Manager", employee.Grade < 20 →

C_1: employee.Sal > 45K, employee.Grade < 18 →

The new constraint C is redundant if the following is unsatisfiable:
(Note: we drop the relation prefix "employee" just for improved readability).

¬(Sal > 40K, Job = "Manager"),
¬(Job ≠ "Manager", Grade < 20),
(Sal > 45K, Grade < 18)

This can be simplified by eliminating the predicates appearing in the negated conjunctions which have stronger ones in the non-negated conjunction. (Eg: sal>45K → sal>40K). The result is:

¬(Job = "Manager"),
¬(Job ≠ "Manager", Grade < 20),
(Sal < 45K, Grade < 18)

This is the same as:

(Job ≠ "Manager"),
(Job = "Manager" OR Grade ≥ 20),
(Sal > 45K, Grade < 18)

which is:

(Job≠"Manager", Job="Manager", Sal>45K, Grade<18) OR
(Job≠"Manager", Grade>20, Sal>45K, Grade<18)

Here, both the conjunctions in the disjunction evaluate to be false, thus making the disjunction unsatisfiable. This indicates that the new constraint is redundant.
Checking unsatisfiability:

Unsatisfiability of the disjunction depends on unsatisfiability of its conjunctions. Here we sketch a graph structure for checking the unsatisfiability of a conjunction.

The conjunction is represented by a directed graph $G = (V,E)$. The vertex set $V$ represents the variables and constants present in the conjunction. Note that we use distinct vertices for different constants belonging to the same domain. The edge set $E$ represents explicit as well as implicit comparisons. Explicit comparisons are the ones present in the conjunction. Implicit comparisons are between constants of the same domain.

The edges are labelled by their comparison operator. As discussed in the previous section, the operators are restricted to $>$, $\geq$, and $\neq$. The "$>$" and "$\geq$" edges are directed, their direction representing the direction of the comparison operator, whereas the "$\neq$" are undirected. Distinct vertices are used to represent different constants even though they belong to the same domain. This representation eliminates the need to assign weights to the edges as in [Rohu80, Jark84].
A directed path from vertex u to vertex v is sequence of directed edges \( e_1, \ldots, e_k, k \geq 1 \), such that there exists a corresponding sequence of vertices \( v_0, v_1, \ldots, v_p \) \( (u = v_0, v = v_p) \) satisfying \( e_k = (v_{k-1}, v_k) \), for \( 0 < k \leq p \).

The graph representing the first conjunction of the above example is presented below. Note that the constants \((18, 45)\) are represented by distinct vertices. Also, note the (implicit) edges between the vertices "18" and "45".

![Graph representation](image)

Fig 8.1: A conjunction graph

**Lemma 8.9:** The (conjunction) graph is unsatisfiable iff any of the following conditions is true:

1. For any "\( \Rightarrow \)" edge, say from vertex u to vertex u, there is a directed path from vertex v to vertex u.
2. For any \("\approx\) edge, say between the vertices \(u\) and \(v\), equality between \(u\) and \(v\) is implied by the conjunction, by a directed cycle of \("\geq\) edges involving \(u\) and \(v\).

**Proof:**

It is obvious that any of these conditions is sufficient to imply unsatisfiability. We proceed to prove the necessity as follows.

A graph is satisfiable iff a satisfactory assignment can be obtained for its vertices. One way of obtaining a satisfactory assignment is by first converting the graph to its transitive reduction, as in chapter 5. We show that obtaining an acyclic transitive reduction is possible iff none of the above condition is true.

The first step in obtaining transitive reduction is grouping the vertices into equivalence classes. Two distinct vertices \(u, v\) belong to an equivalence class if there is an equality-implying cycle incident on them. All the vertices belonging to an equivalence class are assumed to have equality among them. The second condition above violates this assumption, as it presents a \(\approx\)-edge between two such vertices.
The equivalence classes formed above are mapped into super vertices and any parallel edges between super vertices are replaced by single edges (lemma 5.1). The construction of transitive reduction assumes that the graph thus obtained is acyclic. The first condition above violates this assumption by retaining the >-cycles, as those cycles are not equality-implying ones.

If the graph is acyclic (none of the above two conditions being true), a transitive closure of the graph is obtained (lemma 5.2), and then transitively redundant edges are removed from the transitive closure to obtain transitive reduction. As this transitive reduction is acyclic, there exists a topological sort of its super vertices over the edges, by considering the "->" and "->" edges to be the same and neglecting any "≠" edges. The number of different labels required by such a topological sort is same as the number of distinct super vertices in the transitive reduction. The labels thus obtained can then be mapped from the super vertices to the vertices as follows. All the vertices belonging to a super vertex are given the same label as the super vertex, as this satisfies the equality among those vertices. This labelling automatically satisfies the "≠" edges too, as no distinct super vertices has the same label. Thus, the labels represent a
satisfactory assignment to the conjunction, and such a labelling is possible iff none of the conditions of the lemma are true. \( \Xi \)

Verification of both the conditions in the above lemma basically requires verifying reachability of a specific vertex from another one using selected types of edges. This can be achieved by finding transitive closure or transitive reduction [YuOz84], both of which take time proportional to \( O(n^3) \) where \( n \) is the number of vertices in the graph.

A related method is discussed in [RoHu80]. In that paper, (theorems 21, 22; p. 70), a proof is sketched to show that satisfiability of conjunction is NP complete when (and only when) "\( \neq \)" comparison is allowed between the variables. In that proof, satisfiability of the subgraph containing only "\( \neq \)" edges is reduced to \( k \)-colorability, thus concluding the NP completeness. However, reduction to \( k \)-colorability holds only if the cardinality of the domain of the variables is less than the number of variables in the subgraph - an assumption which is rarely true in any practical situation. If the cardinality of the domain is greater than or equal to the number of variables in the subgraph, colorability (and hence satisfiability) of the subgraph becomes polynomial - by assigning all different colors to different vertices, since we have sufficient number of different colors.
Satisfiability of general conjunctive predicates (with inequality comparisons) is shown to be polynomial in [Klug88], when the size of the domain of attributes is greater than the number of variables used in the query for that domain. [SuKL89] discusses various cases of implication problem by converting them into satisfiability problem, based on the above-mentioned results from [RoHu80].

8.3 Maintenance of aggregate constraints

8.3.1 Interaction of simple and aggregate constraints

The interaction between simple implication constraints and aggregate implication constraints can be obtained as follows. For any simple implication constraint of the form \( \Pi_j r.B_j \, op_j \, k_j \rightarrow r.B_q \, op_q \, k_q \), it can be observed that \( \sigma(\Pi_j r.B_j \, op_j \, k_j) \subseteq \sigma(r.B_q \, op_q \, k_q) \). Since the superset preserves the aggregate bounds in many cases, and the aggregate implication constraints specify aggregate values, interaction of simple and aggregate implication constraints can be used to derive aggregate bounds. As an example, a maximum value specified on a subset specifies a lower bound for the maximum on its superset, and a minimum value specified on a subset specifies an upper bound for the minimum on its superset. The same principle can be used to derive bounds on aggregate values like \( \text{count}(u) \), \( \text{sum}(u) \) etc.
To illustrate the interaction between simple constraints and aggregate constraints, let us consider the following examples:

Example 1:
Simple Constraint:

\[ \text{employee}\_\text{Sal}>40K \rightarrow \text{employee}\_\text{Job}=\text{'Manager'}. \]

Aggregate constraint:

\[ \sigma(\text{employee}\_\text{Sal}>45K) \mid \text{count(employee}\_\text{Ssn}) = 140. \]

The set of values represented by the antecedent clause of the simple constraint, \( \sigma(\text{employee}\_\text{Sal} > 40K) \), is a superset of the set of values represented by the qualification clause of the aggregate constraint, \( \sigma(\text{employee}\_\text{Sal} > 45K) \). Since the "count" aggregate specified on a set can be used to specify a lower bound on the "count" aggregate on its superset, we get a new constraint as below. Note that the comparison operator is changed from "=" to ">=" in the aggregate specification.

Derived constraint:

\[ \sigma(\text{employee}=\text{'Manager'}) \mid \text{count(employee}\_\text{Ssn}) \geq 140. \]

Example 2:
Simple constraint:

\[ \text{employee}\_\text{Job}=\text{'Manager'} \rightarrow \text{employee}\_\text{Grade} \geq 20. \]

Aggregate constraint:
\[ \sigma(\text{employee.Job} = '\text{Manager}') \mid \min(\text{employee.Age}) = 45. \]

By a similar reasoning as in the above example, we get a new constraint as below. Note that the aggregate comparison operator is changed from "=" to "\leq".

Derived constraint:
\[ \sigma(\text{employee.Grade} \geq 20) \mid \min(\text{employee.Age}) \leq 45. \]

**Example 3:**

Simple constraint:
\[ \text{employee.Job} = '\text{Manager}' \rightarrow \text{employee.Grade} \geq 20. \]

Aggregate constraint:
\[ \sigma(\text{employee.Job} = '\text{Manager}', \text{employee.Dept} = '\text{Sales}' \mid \sum(\text{employee.Experience}) = 50. \]

The derived constraint is shown below where the comparison operator of the aggregate specification is changed from "=" to "\geq".

Derived constraint:
\[ \sigma(\text{employee.Grade} \geq 20) \mid \sum(\text{employee.Experience}) \geq 50. \]

**Example 4:**

Simple constraint:
\[ \text{employee.Job} = '\text{Manager}' \rightarrow \text{employee.Grade} \geq 20 \]

Aggregate constraint:
\[ \sigma(\text{employee.Job} = '\text{Manager}') \mid \text{avg}(\text{employee.Age}) = 50. \]
This represents a case where a superset does not preserve an aggregate bound. Since the average value of an attribute in a superset can be less than, equal to, or greater than its average value in a subset, no derivation of any new constraint is possible in such a case.

8.3.2 Generalization of Derivation

The interaction between a simple constraint and an aggregate constraint can be generalized in the following context. Consider:
1) a simple constraint: \( \Pi_j r. B_j \text{ op}_j k_j \rightarrow r. B_q \text{ op}_q k_q \) and
2) an aggregate constraint: \( \sigma(\Pi_i r. A_i \text{ op}_i k_i) \mid \text{Agg}(r. A_m) \text{ op}_m k_m \)
such that
3) for each \( j \), \( \Pi_i r. A_i \text{ op}_i k_i \rightarrow r. B_j \text{ op}_j k_j \).

It can be observed from (1)-(3) above that:
\[
\sigma(\Pi_i r. A_i \text{ op}_i k_i) \subseteq \sigma(\Pi_j r. B_j \text{ op}_j k_j) \subseteq \sigma(r. B_q \text{ op}_q k_q).
\]
So, the aggregate value \( \text{Agg}(r. A_m) \text{ op}_m q_m \) specified on \( \sigma(\Pi_i r. A_i \text{ op}_i k_i) \) can be used to derive an aggregate bound on \( \sigma(r. B_q \text{ op}_q k_q) \) to get the aggregate constraint "\( \sigma(r. B_q \text{ op}_q k_q) \mid \text{Agg}(r. A_m) \text{ op}_m' k_m' \)."

The operator \( \text{op}_m' \) depends on the aggregate \( \text{Agg} \) as well as the operator \( \text{op}_m \) and can be used derive bounds on \( \text{Agg}(r. A_m) \) over the
new set of tuples $\sigma(r.A_q \ op_q \ k_q)$, with a few exceptions. The
exceptions are the aggregate $\text{Avg}(u)$ and the operator (op\textsubscript{m}) "m".
The following matrix consolidates the derivations. The entries
where a derivation is not possible are left blank.

<table>
<thead>
<tr>
<th>\textbf{Agg}</th>
<th>$-$</th>
<th>$*$</th>
<th>$&gt;$</th>
<th>$&gt;$</th>
<th>$&lt;$</th>
<th>$&lt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{min}</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>\text{max}</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>\text{avg}(u)</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>\text{sum}(u)</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>\text{count}(u)</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

\textbf{Fig 8.2. Derivation of op\textsubscript{m}' from Agg and op\textsubscript{m}}

Initially, in the aggregate constraint "$\sigma(\Pi_i (r.A_i \ op_i \ k_i)) \mid \text{Agg}(r.A_m) \ op_m \ k_m$", we assumed a restriction on the domain of the attribute $A_m$ to have only positive values. This assumption was to ensure that the aggregate value $\text{Agg}$ on the attribute $A_m$ monotonically increases with the number of tuples in the underlying view. However, it can be observed that the monotonicity is ensured even when this assumption is relaxed in cases where a derivation is possible in the matrix above, except for the aggregate "sum(u)". For the aggregate "sum(u)", the monotonicity is ensured iff the domain of $A_m$ has only positive
values. So we conclude that the restriction on the domain of $A_m$ can be relaxed in the cases of min, max, and count.

8.3.3 Maintenance of aggregate constraints

The maintenance of aggregate constraints ensures that no redundant or contradicting aggregate constraint is added to the set. An aggregate constraint is said to be redundant if it can be derived from the existing set of aggregate and implication constraints. An aggregate constraint is said to be in contradiction if it cannot meaningfully co-exist with the existing constraint set.

To analyze the redundancy and contradiction of aggregate constraints, let us consider the following two constraints with the same aggregate function specified on the same qualifying set of tuples:

a) $\sigma(\Pi_i r.A_i \text{ op } k_i) \mid \text{Agg}(r.A_m) \text{ op } k_{m1}$
b) $\sigma(\Pi_i r.A_i \text{ op } k_i) \mid \text{Agg}(r.A_m) \text{ op } k_{m2}$

Let $x$ be a variable denoting $\text{Agg}(r.A_m)$. We consider four different cases:
1) \( \sigma(x \ op_{m_1} k_{m_1}) \) and \( \sigma(x \ op_{m_2} k_{m_2}) \) are non over-lapping. In this case, the constraints (a) and (b) are in contradiction. For illustration, let the aggregate specifications be "sum(e.Sal) < 1 Million" and "sum(e.Sal) > 2 Million". Obviously, the constraints cannot meaningfully co-exist.

2) \( \sigma(x \ op_{m_1} k_{m_1}) \subseteq \sigma(x \ op_{m_2} k_{m_2}) \). Here, constraint (b) is made redundant by the constraint (a). For example, let the aggregate specifications be "sum(e.Sal) < 1 Million" and "sum(e.Sal) < 2 Million". As the first constraint defines a tighter bound on "sum(e.Sal)", the second constraint becomes redundant.

3) \( \sigma(x \ op_{m_1} k_{m_1}) \supset \sigma(x \ op_{m_2} k_{m_2}) \). As above, constraint (a) is made redundant by the constraint (b).

4) \( \sigma(x \ op_{m_1} k_{m_1}) \) and \( \sigma(x \ op_{m_2} k_{m_2}) \) are over-lapping, but neither is a superset of the other. In this case, constraints have neither redundancy nor contradiction. As an illustration, let the aggregate specifications be "sum(Sal) > 1 Million" and sum(Sal) < 2 Million". This defines a lower and upper bound for the specified aggregate.

The maintenance algorithm, in the context of a new constraint, checks if the existing set can derive another
constraint with the above properties. If the new constraint is redundant or contradicting, it is not accepted to the set. Whenever a new aggregate constraint is accepted, all the existing aggregate constraints on the same qualifying set of tuples are individually checked for redundancy and removed if found redundant.
Chapter 9

CONCLUSION AND FUTURE EXTENSIONS

In this work we have proposed and described a scheme for utilizing semantic constraints for optimizing a database query. We have analyzed three different types of semantic constraints - subset constraints, implication constraints, and aggregate constraints. We have tried to quantify the factors that decide the profit of a query and illustrated how relations, rules, and query can interact to arrive at an optimum query form. The major contribution of this work is a scheme that dynamically selects from a large collection of rules only the profitable ones for a relation in a query context.

Our work differs from the existing related ones in terms of generalized rule representation, dynamic interaction of objects, incorporation of aggregate constraints, and comprehensive semantic maintenance.

An algorithm is introduced to transform the initial query to a semantically equivalent one. The algorithm has its best performance if the estimations of selection-join sequences holds good in reality. Certain major factors like elimination of redundant joins are independent of these assumptions, anyway.
Cases where a query can be answered just using semantic rules and the ones where query conditions and/or semantic constraints imply a null answer are also handled efficiently by the algorithm. In other cases, semantic rules aid the query processing by generating useful additional constraints or by eliminating existing redundant constraints.

The algorithm is implemented with necessary user interface modules and tested with real data of reasonable volume. The test results are very encouraging, thus revealing the potential savings a semantic optimizer can provide. If semantic optimization is treated as a pre-cursor of conventional syntactic optimization, the entire scheme expands the possible spectrum of equivalent forms of the specified query.

In some cases, it is true that the cost of the semantic optimization is not justified by the savings offered by the optimization of the query. But, as the data size increases, the cost of semantic optimization remains unaltered, whereas the savings due to optimization increases. Also, like in most practical cases, with repeated executions of the optimized query, the optimization cost can be amortized over those executions.
A set of algorithms is devised to maintain different types of semantic constraints. Related maintenance schemes assure that the semantic rule set is free from contradiction and redundancy. When a new constraint is added to the existing set, it is checked for semantic soundness.

We are currently studying some additional types of constraints and optimization strategies to incorporate in the algorithm. Usage of conventional constraints like functional dependencies along with the semantic constraints requires further analysis. Methods like introducing an additional join to the original query (join introduction) as an optimizing scheme [King81] also needs further investigation from an efficiency point of view.

Two possible future extensions to the system currently under investigation are semantic categorization and partial optimization. Both these extensions are more relevant to programmed (repetitive) queries rather than interactive ones. Statistically, more than 80% of all the database queries are preprogrammed and highly repetitive. In such a situation, it is quite sensible to optimize the queries only once and save the optimized form for all future uses. But such saved forms are valid only when the relevant semantics (that used for
optimization) remains unchanged. Any change in the semantics mandates a reprocessing of those queries that used the changed semantics for optimization.

In semantic categorization, semantic rules are given weights according to their volatility. This categorization is highly dependent on the nature of data and represents only an approximate stability of different rules. The assumption is that if a rule is more volatile, there are more chances for an optimized query form that used the rule to become invalid within a given time frame, thus mandating a reoptimization. If the rules are categorized, the semantic optimizer can then analyze them before using in an optimization, in terms of profitability and volatility. These factors can be weighed against each other to arrive at an appropriate selection of optimization rules.

The second extension, partial optimization, becomes useful in the case where semantic information used to optimize a query is subjected to change. In such a situation, we are studying the possibilities of avoiding a total reprocessing of the user specified query. Reflecting the semantic changes into the previously processed queries without reprocessing them from the original form could be profitable especially if semantic processing is costly.
Also, we are conducting more experiments using the optimizer with various practical situations. Our studies involve different table sizes, various storage structures and secondary indices, and different query patterns.
REFERENCES


[HaZd80]: Hammer M., and Zdonik S B Jr., Knowledge based query processing, VLDB 1980, 137-147.


[JaCV84]: Jarke M., Clifford J., and Vassiliou Y., An optimizing Prolog front end to a relational query system, Proc. ACM SIGMOD, Boston 1984, pp.296-306.


[ShOz89]: Shenoy S T., Ozsoyoglu Z M., Design and implementation of a semantic query optimizer, IEEE Transactions on Knowledge and Data Engineering, September 1989.

[ShSD88]: Shekar S., Srivastava J., Dutta S., A formal model of trade-off between optimization and execution costs in


[YuOz84]: Yu T C., and Ozsoyoglu Z M., On determining tree query membership of a distributed query, INFOR Aug 1983, 261-281