Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Techniques for solving constraint satisfaction problems (CSP) have progressed independently of work on relational databases (RDB) even though intuitively, the benefits of integrating these two fundamental areas have always been apparent. As opposed to just using the RDB as a data storage back end, an integrated approach enables a user to specify, solve, and present CSP solutions *within the database environment*. Important advantages include: 1) The user works within a single programming language. There is no need to switch programming environment and to confront the impedance mismatch associated with embedded database programming. 2) There is a strong integration between constraint modeling and data definition. A database design also serves as its constraint specification. 3) There are well-established practices and tools for relational databases that can further support the process of constraint

modeling. Examples are Entity-Relationship Diagrams for specifying constraints, and Query-By-Example for developing SQL queries.

Early attempts to marry CSP and RDB include *deductive databases* which reuse important ideas from *logic programming*. The limited success of deductive databases in industrial applications reveals an important lesson: it is not easy to supplant a popular language such as the Structured Query Language (SQL) regardless of the attributes of the contender.

Given this lesson, a natural question is to ask if SQL, and database techniques in general, can be leveraged for modeling and solving constraints. Recent work by Cadoli and Mancini shows that SQL is sufficiently expressive for constraint specification, but leaves unanswered the following questions. (1) How can such a specification be efficiently solved? (2) What extensions to the language best suit the large and varied SQL user base present in industry today? Thus there remains a large gap in bringing SQL CSPs to the RDB community. Our research attempts to narrow this gap by proposing RDB extensions and techniques that enable typical RDB practitioners to solve CSPs over relational data in an intuitive and efficient manner. We begin by formally defining the related concepts.

1.2 Constraint Satisfaction Problem

Definition 1. A *constraint satisfaction problem* [26, 2] (CSP) is a pair $(\mathcal{V}, \mathcal{C})$ where

1. $\mathcal{V} = \{V_1, \ldots, V_n\}$ is a finite set of *variables*. For each variable $V_i \in \mathcal{V}$ a finite domain of *values*, $Dom(X) = \{x_1, \ldots, x_j\}$
2. $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a finite set of *constraints*. Each constraint C_i is a a pair (P_i, r_i), where P_i is a list of variables of length m_i and r_i is a m_i-ary relation over P_i. The tuples of r_i indicate the allowed or forbidden combinations of values for the variables P_i.

Definition 2. A *solution* to a CSP is a set of assignments to all n variables that satisfies all k constraints.

1.3 Relational Database

Assume a finite set U of attributes, where an attribute $A \in U$ has an associated finite set of values, $\Delta(A)$, called A's domain.

Definition 3. A *relation schema* $R = \{A_1, \ldots, A_n\}$ is a subset of attributes in U.

Definition 4. A *relation* r over a relation schema R is a finite set of R-tuples, where each R-tuple is a mapping from each element A of R to a value in $\Delta(A)$.

Definition 5. A *relational database* is a pair (D, d) where

1. D is a *database schema* consisting of a finite set of relation schemas,
2. d is a database consisting of a set of relations, one for each relation schema R of D.

Sometimes we use the familiar notation $R(A_1, ..., A_n)$ to represent the relation schema $R = \{A_1, ..., A_n\}$. We denote the domain of an R-tuple as $\Delta(R) = \Delta(A_1) \times \ldots \times \Delta(A_n)$. The size, $|\Delta(R)|$, of $\Delta(R)$ is $|\Delta(A_1)| \times \ldots \times |\Delta(A_n)|$, where $|\Delta(A_i)|$ is

the size of $\Delta(A_i)$. Given a relation r, $\alpha(r)$ denotes the relation schema of r. If t is an X-tuple, and $Z \subseteq X$, then $t[Z]$ indicates the restriction of t to the attributes Z.

Definition 6. Given sets of attributes $X, Y \in R$, $X \rightarrow Y$ specifies a *functional dependency* of Y on X over R. The dependency requires that for any relation r over R, whenever $t_1[X] = t_2[X]$ for R-tuples t_1 and t_2 in r, then $t_1.Y = t_2.Y$. Clearly, if Y contains all attributes of R, then $X \rightarrow Y$ implies that $t_1[X] \neq t_2[X]$, for any $t_1 \neq t_2$. The set X is known as a *key* of R.

1.4 Relational Algebra

The Relational Algebra (RA) is the mathematical foundation for relational database querying. We describe the notation for relational algebra from ref[Kanellakis].

A relational algebra expression E over a database schema $D = \{R_1, ..., R_m\}$ is defined as follows.

Definition 7. $\Pi_X(r)$ is the *projection* of relation r onto relation schema X.

1. $X \subseteq \alpha(r)$ and $\alpha(\Pi_X(r)) = X$
2. $\Pi_X(r) = \{t[X] | t \in R\}$

Definition 8. $r_1 \bowtie r_2$ is the (natural) *join* of r_1 and r_2.

1. $\alpha(r_1 \bowtie r_2) = \alpha(r_1) \cup \alpha(r_2)$
2. $r_1 \bowtie r_2 = \{t | t$ is an $\alpha(r_1) \cup \alpha(r_2)$-tuple,
   such that $t[\alpha(r_1)] \in r_1$ and $t[\alpha(r_2)] \in r_2$

Definition 9. $\sigma_{A=B}(r)$ is the *selection* on r by $A = B$.

1. $A, B \in \alpha(r)$ and $\alpha(\sigma_{A=B}(r)) = \alpha(r)$

2. $\sigma_{A=B}(r) = \{t | t \in R \text{ and } t[A] = t[B]\}$

Definition 10. $\rho_{B|A}(r)$ is the *renaming* in r of A into B.

1. $A \in \alpha(r), B \notin \alpha(r)$ and $\alpha(\rho_{B|A}(r)) = (\alpha(r) - \{A\}) \cup \{B\}$
2. $\rho_{B|A}(r) = \{t | \text{ for some } t' \in r, t[B] = t'[A] \text{ and } t[C] = t'[C] \text{ when } C \neq B\}$

The schema of E, $\alpha(E)$, can be determined from the schema restrictions of the operations. Given an expression E and a database d, $E(d)$, or simply $E()$ when d is understood, is the relation obtained by evaluating E on d.

1.5 Relational CSP

A *relational constraint satisfaction problem* (RCSP) is a CSP defined over a relational database where constraints are expressions of the form $E = \emptyset$ or $E \neq \emptyset$ where E is a relational algebra expression. A constraint C of type $E = \emptyset$ or $E \neq \emptyset$ is said to *hold on a database* d if $E(d) = \emptyset$ or $E(d) \neq \emptyset$ respectively.

Definition 11. A RCSP Ξ over a database schema D is a pair $(\mathcal{S}, \mathcal{C})$ such that

1. $\mathcal{S} = \{S_1, \ldots, S_m\}$, is a finite set of relation schemas[1],
2. $\mathcal{C}$ is a finite collection of constraints over $\mathcal{S} \cup D$.

Definition 12. A *solution* to Ξ on a database d of D, is a database $\mathbf{s}$ of $\mathcal{S}$ such that each constraint in $\mathcal{C}$ holds on the database $d \cup \mathbf{s}$.

[1]Note that $\mathcal{S}$ are the *guess* relations of [4]

We call elements of $\mathcal{S}$ *constraint relation schema* (or simply *constraint schema*) and relations over constraint schemas *constraint relations*.

Note that unlike traditional CSP (of Definition 1), RCSP does not have an explicit notion of variables; variables can be interpreted as the possible tuples of the constraint relations in $\mathcal{S}$. For each $S \in \mathcal{S}$, an S-tuple corresponds to an element in $\Delta(S)$, the domain of S. The constraints in $\mathcal{C}$ limit which combinations of S-tuples may be simultaneously present in the *solution database* $\mathbf{s}$. Once $\mathbf{s}$ is found, we can interpret a variable as $true$ or $false$ depending on whether or not the corresponding S-tuple is present in $\mathbf{s}$. As $\mathbf{s}$ corresponds to a set of assignments to all variables that is consistent with all constraints, $\mathbf{s}$ is a solution to the CSP.

We give a few more useful notations. Let E be a relational algebra expression.

- $\phi(E)$ is the set of constraint schemas that appear in E.
- $\mu(E)$ are the the set of attributes that appear in E.
- $\mu(E|X)$ is $\mu(E) \cap X$, where $X \subseteq U$.
- $E\{M'_1/M_1, ..., M'_k/M_k\}$ is the expression obtained by replacing M'_i for M_i, $1 \leq i \leq k$, wherever M_i occurs in E. M_i, M'_i are RA-expressions.

1.6 SQL CSP

In general, the SQL CSP is a CSP expressed in SQL which matches the RCSP definition $\Xi = (\mathcal{S}, \mathcal{C})$ over some database schema D, where $\mathcal{S}$ is a database schema, and $\mathcal{C}$ is a set of constraints. As SQL is based on Relational Algebra, it includes a Data Definition Language (DDL) capable of expressing relational schema $\mathcal{S}$ and D. Each schema is composed of attributes with specified domains. Each constraint in $\mathcal{C}$ can be expressed using SQL's `EXIST` (i.e. $E \neq \emptyset$) or `NOT EXIST` (i.e. $E = \emptyset$) operators preceding a select query of the form `SELECT...FROM...WHERE...` which evaluates to a set of tuples from $\mathcal{S}$ that are either allowed or forbidden to be simultaneously present in the final solution. Other SQL patterns can also be used to express collections of constraints succinctly.

1.7 Example Use Case

We give the graph coloring problem as an example RCSP specification. The problem requires each element of a set of nodes $\mathcal{N} = \{N_1, \ldots, N_n\}$ to be assigned to a color in a set of colors $\mathcal{K} = \{K_1, \ldots, K_k\}$, with the constraint that no two adjacent nodes (nodes linked by an edge) be assigned the same color. The RCSP is given as $\Xi = (\mathcal{S}, \mathcal{C})$ over database schema D where:

- D is a set of three relation schemas Colors(cid), Nodes(nid),

and Edges(n1,n2) describing the colors, nodes, and edges of the problem.

- $\mathcal{S}$ contains the single relational schema $Coloring(nid, cid)$.
- and $\mathcal{C}$ contains the constraints necessary to enforce that:
  - each node is assigned exactly 1 color:

    $\sigma_{nid=i}(Coloring) \neq \emptyset$ for all $i \in \Delta(nid)$,

    $\sigma_{c1.nid=c2.nid \wedge c1.cid \neq c2.cid}(\rho_{c1}(Coloring) \times \rho_{c2}(Coloring)) = \emptyset$
  - nodes connected by an edge do not get mapped to the same color:

    $\sigma_{c1.nid=e.n1 \wedge c2.nid=e.n2 \wedge c1.cid=c2.cid}($

    $\rho_{c1}(Coloring) \times \rho_{c2}(Coloring) \times \rho_e(Edges)) = \emptyset$

Intuitively, a constraint $C_i \in \mathcal{C}$ is represented by a query returning allowed or forbidden simultaneous combinations of tuples in $\mathcal{S}$. For example such a query might return the set of tuples $\{n1 : red, n2 : red\}$ representing that nodes 1 and 2 may not both be colored red (assuming they share an edge). As a result, the relation s is only a valid solution if it dose not contain both tuples simultaneously. In practice, RCSP systems allow expressive constraint queries enabling problems like graph coloring to be specified in 1-2 SQL statements.

Chapter 2

Related Work

In this chapter we discuss related work. We begin by describing research specifically related to SQL CSPs, then we discuss Deductive Databases to highlight previous work that attempted to merge database and constraint programming outside of the relational model.

2.1 CONSQL

CONSQL [4] is a research effort recently proposed by Cadoli and Mancini, in which SQL and its relational algebraic foundation are adopted as the basis for expressing constraints over relational data. The key concept is the non-deterministic GUESS operator that declares a set of relations to have an arbitrary extension. A set of constraints, written in usual SQL syntax over both existing and guess relations, specifies conditions that extensions to the guess relations must satisfy. The work lays down the theoretical foundation for leveraging SQL as a CSP specification language. Below is an example encoding of the graph coloring problem described in Section 1.7.

```
CREATE SPECIFICATION Graph_Coloring (
  GUESS TABLE COLORING AS
    SELECT n, color FROM TOTAL FUNCTION_TO(COLORS) AS color OF NODES
      CHECK ( NOT EXISTS (
      SELECT * FROM COLORING C1, COLORING C2, EDGES
      WHERE C1.n <> C2.n AND C1.color = C2.color
        AND C1.n = EDGES.f AND C2.n = EDGES.t ))
      RETURN TABLE SOLUTION AS SELECT COLORING.n, COLORS.name
        FROM COLORING, COLORS WHERE COLORING.color = COLORS.id
)
```

The `SPECIFICATION` keyword signals a CSP, whose `GUESS` table is the relation `COLORING(node,color)`, where `node` and `color` are values taken from two given relations, `NODES` and `COLOR`, respectively. A valid instance of `COLORING` must be a total function from `NODES` to `COLORS`. The only constraint of the problem is described by the `CHECK NOT EXISTS` statement, which assumes a third relation, `EDGE`, and asserts that there must not be two adjacent nodes colored by the color. Finally, the returned solution is a view over the `COLORING` and `COLORS` relations that displays, for each node, the associated color name.

Cadoli and Mancini also propose a prototype solver which employs various local (i.e., incomplete) search techniques to generate candidate solutions. Each candidate is checked independently by the RDB engine for consistency with the constraints.

2.2 D-Wave

Dwave [5] is a company that researches adiabatic quantum computing in the context of solving complex search and optimization problems. They have created a SQL extension for CSPs called SirQL [6] which, according to a brief conversation with their engineers in August 2009, relies on traditional CSP solvers for constraint solving. With little information available, we can only surmise that they have a similar translation system to SCDE, and we cite this reference mainly to support the notion that industry is interested in SQL CSPs. Below is an example encoding in SirQL of the graph coloring problem described in Section 1.7.

```
FIND Coloring (vtx UNIQUE COMPLETE, col)
FROM Vertex, Color
WHERE NOT EXISTS (SELECT *
  FROM Coloring cg1, Coloring cg2, Edge e
  WHERE cg1.vtx = e.vtx1 AND cg2.vtx = e.vtx2
  AND cg1.col = cg2.col)
```

This encoding is very similar to CONSQL.

2.3 Deductive Databases

Deductive databases [7] are databases that support declarative rule-based queries which can be used to infer (or deduce) new facts based on existing facts and rules

stored in the database. The reasoning power of deductive databases is based on logic programming semantics, and usually combines both forward and backward chaining techniques. Deductive databases have grown out of the desire to overcome the "impedance mismatch" of using procedural languages with relational database languages like SQL. Below is an example encoding for (disjunctive) deductive databases [8] of the graph coloring problem described in Section 1.7.

$$col(X, r) \vee col(X, g) \vee col(X, b) \leftarrow node(X).$$
$$\leftarrow edge(X, Y), col(X, C), col(Y, C).$$

The first rule references a set of facts (`node(X)`) in the database representing the nodes of the graph. It uses the '$\leftarrow$' (implication) operator to enforce that each node receive a color ('r', 'g', 'b'). The second rule references a set of facts (`edge(X,Y)`) representing the edges of the graph. It implies that no two nodes sharing an edge may have the same color.

Aside from the language obstacles mentioned in the introduction, deductive databases have also been hampered by efficiency issues in their implementations, which have mostly relied on resolution-based inference systems that do not scale well. Promising recent work on answer-set programming adopt techniques closer to SAT solvers.

2.4 Open Research Quesions

To summarize, the main weaknesses of existing systems are:

1. **Usability.** Languages that are expressive and succinct, such as Deductive Databases, have failed to receive widespread acceptance due to intimidating mathematical prerequisites and lack of cohesion with existing RDB techniques.

2. **Scalability.** Efforts more closely aligned with RDB have so far not been able to scale effectively, especially for problems with many variables and large variable domains.

Our research aims to address parts of these issues by focusing on the following two questions in the context of bringing constraint solving capabilities to RDBs.

- How can we simplify the task of specifying constraint programs in relational databases?

- How can the performance of solving SQL-based constraints be improved to extend the practical reaches of such systems?

Chapter 3

Proposed Solution

Consider the problem of creating a teaching schedule over the database schema described by the ER diagram in Figure 3.1. The constraint schema is the shaded relationship `schedule`. The diagram is annotated with several constraints. First, the total participation (represented by the thick line between `course` and `schedule`) along with the cardinality constraint indicates that each course must be taught exactly once. Second, the functional dependency `professor, time`→`course` specifies that no one may teach two courses at the same time. Other cardinality constraints of note: n represents the number of courses a professor must teach (specified by the `professor.num` field), while m represents the number of courses that can be taught during one time slot (i.e. the number of rooms available).

The example illustrates how existing industry tools and techniques can serve as a strong starting point for CSP specification. To leverage existing standards optimally, a SQL CSP tool should strive to deviate minimally from existing constraint language syntax. As RDB practitioners have experience with implementing ER diagrams in

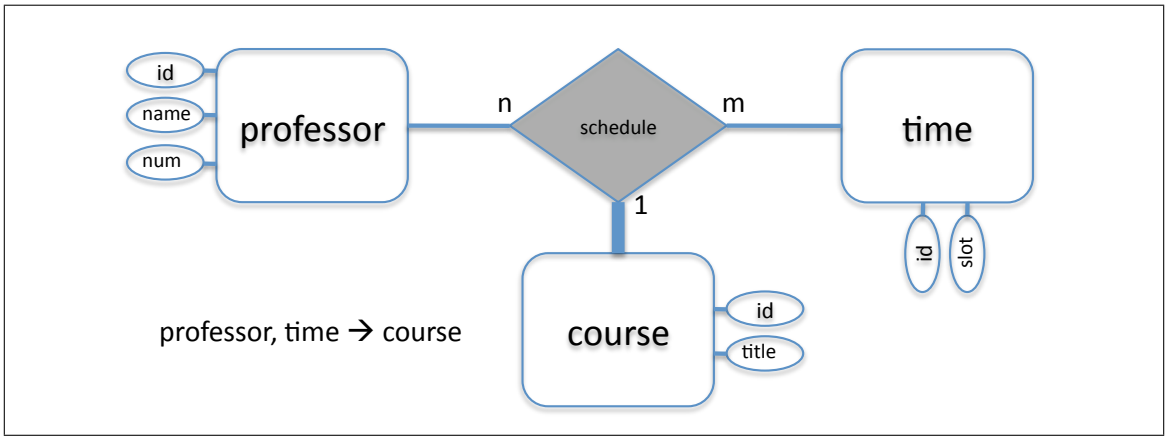

Figure 3.1: Course Scheduling ER Diagram

SQL's Data Definition Language (DDL), this language is a natural choice for RCSP specification. The main challenge our research addresses is how to efficiently solve RCSP specified in SQL's DDL. To this end, we offer the following contributions:

- A specific subset of SQL's DDL useful for RCSP specification
- A set of algorithms for translating RCSP into Boolean Satisfiability problems
- An algorithm for automated problem decomposition
- A technique for enabling varied back-end boolean solving algorithms

3.1 Leveraging SQL's Data Definition Language

One of the complexities of most CSP specification languages is the need to come up with a set of variables, and corresponding domains to model the problem. Previous

CSP SQL efforts have dealt with these requirements in different ways. For example, in CONSQL and DWave users explicitly consider these notions through keyword expressions such as TOTAL FUNCTION_TO and COMPLETE. Both of these expressions (seen in Chapter 2) serve to differentiate variable attributes from domain attributes allowing the solver to automatically enforce that each CSP variable be assigned exactly one value. In contrast, we bypass explicit keywords for traditional CSP concepts and adhere to commonly occurring notions of relational database. This potentially lowers the learning costs for SQL practitioners, and allows them to focus on constraints in the database sense: as requirements on the relationships between attributes of entities. To accomplish this we use several features of SQL's Data Definition Language.

SQL's Data Definition Language (DDL) is an imperative language that uses statements to modify a database schema via adding, changing, or removing elements of relations and other objects. As SQL's DDL is not truly separate from it's query sublanguage; they can be mixed in data definition statements. More importantly, data integrity constraints can be added to any relation schema via the "`CONSTRAINT CHECK`(e)" statement, where e is any boolean SQL expression. Given a SQL DDL schema, we interpret relation schemas augmented with integrity constraints as the schema of *constraint relations* mentioned in Section 1.5. Unlike tradi-

tional database implementations, we allow e to be a complex expression, which may reference any set of relations in the database including other constraint relations. This enables the complete specification of RCSPs within the existing syntax of SQL.

Assuming that a database system designed to process such RCSPs contains the database and its schema, the user need only specify the names of the relations to solve. The system can interpret the schema as an RCSP $\Xi = (\mathcal{S}, \mathcal{C})$ over D, where $\mathcal{S}$ is the set of relation schemas specified by the user, C is the set of data integrity constraints belonging to the relation schemas in S, and D is all other relation schemas in the database schema. The system can then attempt to solve the problem and, if feasible, automatically populate a constraint relation for each schema in $\mathcal{S}$ with with the corresponding solution tuples.

3.2 Translating RCSP to Boolean Satisfiability

As RCSP do not necessarily specify variables and domains in the traditional CSP sense, it is difficult to directly apply many solving algorithms that have been previously developed. One approach to addressing this issue is to translate a RCSP into a more traditional CSP encoding. As mentioned in Section 1.5 the variables of a RCSP can be thought of as the possible tuples of the relations over $\mathcal{S}$. Each S-tuple can either be present in the solution or not, representable as $true$ or $false$ respectively.

Thus, boolean satisfiability (SAT) is a natural CSP target encoding for RCSP translation. Furthermore, as a well studied CSP language, boolean satisfiability allows us to take advantage of the substantial progress that has been made in the last decade in the sophistication and performance of SAT solvers.

Assume we are given a RCSP $\Xi = (\mathcal{S}, \mathcal{C})$ over database schema D where each constraint $C \in \mathcal{C}$ imposes either allowed or forbidden sets of S-tuples via SQL DDL statements of the form "`CHECK EXISTS`(e)" or "`CHECK NOT EXISTS`(e)" respectively (see Section 1.6). As these constraints are originally written as data integrity constraints, e is typically a SQL `SELECT` query which references one or more relation schemas $S \in \mathcal{S}$ and zero or more relation schemas in D. The schema of e, $\alpha(e)$, is the set of attributes from $\mathcal{S} \cup D$ that results from executing e. In order to translate the constraints into boolean formulae, two steps are necessary.

Step 1. Since e references constraint schemas in $\mathcal{S}$ that do not yet have associated relations, it must be converted to e' such that the result of executing e' is a set of boolean variables which can be mapped to possible S-tuples.

Step 2. The boolean variables must be organized into boolean clauses enforcing the allowed or forbidden relationships established by the original constraint C.

In order to accomplish the first step, we impose some limitations on the structure of e and introduce *variable maps*. The second step is accomplished by recognizing

the SQL DDL pattern within which e is found, and applying a pattern specific translation algorithm. The output of these algorithms is a boolean formula adhering to an encoding form we call *Extended Conjunctive Normal Form* (eCNF).

The solution to the eCNF formula is an assignment to all variables for which the formula evaluates to *true*. The solution to the original RCSP is a database s of $\mathcal{S}$ created by selecting the tuples for which the corresponding variables are assigned to *true* in the eCNF solution.

3.2.1 Variable Map

Variable maps (VBmaps) are temporary relations defined outside of D or $\mathcal{S}$ that map all possible tuples of $\mathcal{S}$ to unique variable identifiers. For each $S \in \mathcal{S}$ a VBmap $\nu(S)$ is created and populated with $\Delta(S)$. The schema $\alpha(\nu(S))$ is $S \cup \{vbid\}$, where $vbid$ is a unique integer identifying a potential S-tuple.

As mentioned above in step 1, e must be converted to e' before its execution can return useful values. In order to simplify the process, we restrict e to be a query in which relations schemas in $\mathcal{S}$ appear in the `FROM` clause, but not in any nested queries. Note that e may be a nested query. The algorithm $vb(e)$, given in Figure 3.2, uses VBmaps to enable the execution of e' to return a set of variable ids corresponding to possible tuples in $\mathcal{S}$. These variable ids are then interpreted as boolean variables forming clauses in eCNF.

Algorithm $vb(e)$

1. Input: e is an expression (query) referencing constraint schemas $\phi(e) = \{S_1, \ldots, S_k\}$.
2. $e' \leftarrow e\{\alpha(\nu(S_1))/S_1, \ldots, \alpha(\nu(S_k))/S_k\}$
3. $\alpha(e') = \{\alpha(\nu(S_1)).vbid, \ldots, \alpha(\nu(S_k)).vbid\}$
4. return e'

Figure 3.2: VBmap Swap Algorithm

3.2.2 Extended Conjunctive Normal Form

Conjunctive Normal Form (CNF) is the form for boolean satisfiability formulae expected by most solvers. For many interesting constraints, the corresponding CNF boolean encoding is impractically long. For this reason, we propose an Extended Conjunctive Normal Form (eCNF) which generalize CNF by allowing 1)conjunctions of literals to occur in place of literals and 2), a cardinality constraint. Each clause in eCNF has the form $\mathcal{F}$ @ k where $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ is a set of formulae composed of conjunctions of literals, k is a non-negative integer $0 \leq k \leq n$, and @ is an integer comparison from the set $\{>, \geq, <, \leq, =\}$. Such a clause is true if @k formulae among $F_1, ..., F_n$ are true. For example $\{(v_1 \wedge v_2 \wedge v_3), (\neg v_4), (\neg v_5 \wedge v_6)\} \geq 2$ is true if at least two of $(v_1 \wedge v_2 \wedge v_3), (\neg v_4)$ and $(\neg v_5 \wedge v_6)$ are true. Like CNF, an eCNF sentence is a conjunction of these clauses, and thus is only satisfied if all clauses are satisfied.

The eCNF notation succinctly captures some important concepts including car-

dinality constraints, grouping and disjunctive normal form. Lastly, eCNF facilitates easy translation to common constraint representations such as: CNF for SAT solvers, pseudo-boolean constraints [10, 13], and MPS for integer programming solvers.

3.2.3 SQL Constraint Patterns

As mentioned in Chapter 1, RCSP constraints limit constraint relations by specifying combinations of allowed or forbidden S-tuples. This can be accomplished in SQL with the `CHECK EXISTS`(e) and `CHECK NOT EXISTS`(e) constraint patterns mentioned above. For example, consider the constraint of the graph coloring problem that no two nodes connected by an edge may have the same color. This constraint can be specified as a `CHECK NOT EXISTS` constraint where e is the query

```
SELECT *
FROM Coloring cg1, Coloring cg2, Edge e
WHERE cg1.node = e.node1 AND cg2.node = e.node2
  AND cg1.col = cg2.col
```

In order to be processed as a SAT problem, this constraint must be translated into a set of boolean clauses of the form $(\neg v_i \vee \neg v_j)$ where the boolean variables v_i, v_j represent a pair of possible tuples in the constraint relation `Coloring` that corresponds to adjacent nodes assigned to the same color. This is accomplished in two steps as follows. First, the query e is converted into the query e' given below.

```
SELECT cg1.vbid, cg2.vbid
```

```
FROM Vbmap_Coloring cg1, Vbmap_Coloring cg2, Edge e
WHERE cg1.node = e.node1 AND cg2.node = e.node2
  AND cg1.col = cg2.col
```

Second, the query e' is executed and the resulting tuple set T is used to create a set of eCNF clauses of the form $(\neg t[cg1.vbid], \neg t[cg2.vbid]) > 0$ for each tuple $t \in T$.

Although the two SQL patterns, `CHECK EXISTS`(e) and `CHECK NOT EXISTS`(e), are syntactically sufficient for expressing CSPs, they often lead to impractically long problem specifications. Other patterns exist which enable much more succinct expression of complex constraints. These higher level patterns also enable solvers to detect important structural properties[11] of the problem, which can reduce solve time. This is a well studied concept and typical CSP languages incorporate a variety of custom operators and functions[33] to enable these efficiencies. As relational algebra and SQL provide a powerful set of operators that allow complex constraints to be expressed in a wide variety of ways, we do not need to introduce custom syntax. Instead, we face the complement of the problem: how to translate an arbitrary SQL constraint into an "optimal" CSP constraint. This is an open problem. As a starting pint, we choose to limit our SQL constraints to a set of four patterns that we have found to be sufficient for expressing most finite CSPs naturally and, for each pattern, describe a translation algorithm that produces an eCNF encoding.

In each of these patterns there exists a *main query*, e, which is converted into a query e' by the substitution algorithm $vb(e)$ described in Section 3.2.1. The four constraint patterns; `CHECK EXISTS`, `CHECK NOT EXISTS`, `CHECK COUNT`, and `CHECK NOT EXISTS COUNT` are abbreviated with the acronyms CE, CNE, CC, and CNEC respectively.

Note that, given an original SQL constraint C, the validity of the eCNF formula generated by an algorithm can be demonstrated by verifying that:

1. every satisfying variable assignment represents S-tuples of a database $\mathbf{s}$ for which C holds, and

2. every non-satisfying variable assignment represents S-tuples of a database $\mathbf{s}$ for which C does not hold.

In other words, the solutions to the eCNF formula must map exactly to all possible databases $\mathbf{s}$ which satisfy the original constraint.

Check Exists Pattern

The most basic constraint in a CSP lists combinations of simultaneous values (tuples) that are allowed in the solution. At least one combination must be present in the solution for the constraint to be satisfied. For example, the graph coloring problem might be augmented with a constraint forcing nodes 1 and 2 to

have the same color. This would be represented by the tuple combination list: {{(1,red),(2,red)},{(1,greed),(2,green)},{(1,blue),(2,blue)},...}. The corresponding SQL constraint would be

```
CHECK EXISTS ( SELECT *
  FROM Coloring cg1, Coloring cg2
  WHERE cg1.node = 1 AND cg2.node = 2 AND cg1.col = cg2.col)
```

where e is the `SELECT` query.

The general pattern of the constraint is `CHECK EXISTS(e)` where e is the main query. The algorithm for translating this constraint into eCNF is given in Figure 3.3.

Algorithm *checkE*(C)

1. Input: C is a CE constraint, e is the main query of C
2. $e' \leftarrow vb(e)$
3. Return one eCNF clause $\mathcal{F} > 0$
   where for each tuple $t \in e'()$ there is an $F_i \in \mathcal{F}$ such that
   $F_i = (\wedge_{S \in \phi(e)} t[\nu(S).vbid])$

Figure 3.3: Algorithm for CE constraints

The correctness of the algorithm can be explained as follows. In order for a CE SQL constraint C to hold, the query e must return at least one tuple when executed against the solution database $\mathbf{s} \cup d$. Note that the algorithm creates a conjunction F_i for each combination of possible S-tuples from $S \in \phi(e)$ matched by e. As $\mathcal{F} > 0$

requires at least one F_i to be *true*, every satisfying variable assignment to the eCNF formula maps to a unique database $\mathbf{s}$ for which C holds. A non-satisfying assignment implies that corresponding to each F_i, some S-tuple combination is absent in $\mathbf{s}$ and thus C does not hold.

Check Not Exists Pattern

The Check Not Exist constraint is the dual of the CE constraint. In a traditional CSP it lists combinations of simultaneous values (tuples) that are forbidden in the solution. The constraint is satisfied only if the solution contains none of the combinations listed. The example given at the top of Section 3.2.3 would generate a tuple combination list: {{(x,red),(y,red)},{(x,green),(y,green)},{(x,blue),(y,blue)},...} for every pair of adjacent nodes x and y.

The general pattern of the constraint is `CHECK NOT EXISTS(e)` where e is the main query. The algorithm for translating this constraint into eCNF is given in Figure 3.4.

The correctness of the algorithm can be explained as follows. In order for a CNE constraint C to hold, the query e must return 0 tuples when executed against the solution database $\mathbf{s} \cup d$. Note that an eCNF clause is created for each combination of S-tuples for $S \in \phi(e)$ matched by e. As each eCNF clause requires the absence of at least one S-tuple from each combination, a satisfying variable assignment ensures

Algorithm *checkNE*(C)

1. Input: C is a CNE constraint, e is the main query of C
2. $e' \leftarrow vb(e)$
3. Return a set of eCNF clauses such that
   for each $t \in e'()$ there is an eCNF clause $\mathcal{F} > 0$
   with one $F_i \in \mathcal{F}$ for each $S \in \phi(e)$ such that
   $F_i = (\neg t[\nu(S).vbid])$

Figure 3.4: Algorithm for CNE constraints

such combinations will not be returned by e. Furthermore, there exists a satisfying variable assignment for every S-tuple combination that does not match e, and by extension, every possible database s for which C holds.

Check Count Constraint

The CC constraint is a generalization of the CE and CNE constraints. Combinations of simultaneous values (tuples) are given a cardinality comparison constraint, $@k$, where k is a non-negative integer, and @ is an integer comparison from the set $\{>, \geq, <, \leq, =\}$. For example, the graph coloring problem might be augmented with a constraint forcing at least three nodes to be the color 'red'. The corresponding SQL constraint would be

```
CHECK ( 3 <=  ( SELECT COUNT(*)
  FROM Coloring cg1 WHERE cg1.col='red'))
```

where $k = 3$, @ is '=', and e is the `SELECT` query.

The general pattern of the constraint is `CHECK (k @ e)` where e is the main query and returns a count of the result set. The algorithm for translating this constraint into eCNF is given in Figure 3.5.

Algorithm *checkC*(C)

1. Input: C is a CC constraint, e is the main query of C
2. $e' \leftarrow vb(e)$
3. $@' \leftarrow$ translate @ from left hand side to right hand side.
4. Return one eCNF clause $\mathcal{F}@'k$
   where for each each tuple $t \in e'()$ there is an $F_i \in \mathcal{F}$ such that
   $F_i = (\wedge_{S \in \phi(e)} t[\nu(S).vbid])$

Figure 3.5: Algorithm for CC constraints

We demonstrate the validity of the CC algorithm for the two boundary cases $\mathcal{F} > 0$ and $\mathcal{F} = 0$ by showing how the resulting eCNF is equivalent to the CE and CNE algorithms respectively.

Proposition 1. *A CE constraint is logically equivalent to a CC constraint with* @ = '$>$' *and* $k = 0$.

Proof. Let $e'() = \{Q_1, Q_2, \ldots, Q_n\}$ denote the tuple set returned by executing a VBMap query e', where $Q_i = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$, m is the size of $\phi(e)$, and v_{ij} is a boolean variable. Given $e'()$, the CE, and CC algorithms generate identical eCNF

clauses given below.

$$\{(v_{11} \wedge v_{12}, \wedge \ldots \wedge v_{1m}), (v_{21} \wedge v_{22}, \wedge \ldots \wedge v_{2m}), \ldots (v_{n1} \wedge v_{n2}, \wedge \ldots \wedge v_{nm})\} > 0$$

Therefore, the CE constraint is logically equivalent to a CC constraint

with @ =‘ >’ and $k = 0$.

Proposition 2. *A CNE constraint is logically equivalent to a CC constraint with @ =‘ =’ and $k = 0$.*

Proof. Let $e'() = \{Q_1, Q_2, \ldots, Q_n\}$ denote the tuple set returned by executing a VBMap query e', where $Q_i = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$, m is the size of $\phi(e)$, and v_{ij} is a boolean variable. Given $e'()$, the CNE algorithm generates the eCNF clauses given below.

$$\{(\neg v_{11}), (\neg v_{12}), \ldots (\neg v_{1m})\} > 0$$

$$\{(\neg v_{21}), (\neg v_{22}), \ldots (\neg v_{2m})\} > 0$$

$$\ldots$$

$$\{(\neg v_{n1}), (\neg v_{n2}), \ldots (\neg v_{nm})\} > 0$$

The CC algorithm generates the single eCNF clause given below.

$$\{(v_{11} \wedge v_{12}, \wedge \ldots \wedge v_{1m}), (v_{21} \wedge v_{22}, \wedge \ldots \wedge v_{2m}), \ldots (v_{n1} \wedge v_{n2}, \wedge \ldots \wedge v_{nm})\} = 0$$

This is equivalent to the following set of eCNF clauses because in order for $\mathcal{F} = 0$, each $F_i \in \mathcal{F}$ must be false.

$$\{(v_{11} \wedge v_{12} \wedge \ldots \wedge v_{1m})\} = 0$$

$$\{(v_{21} \wedge v_{22} \wedge \ldots \wedge v_{2m})\} = 0$$

$$\ldots$$

$$\{(v_{n1} \wedge v_{n2} \wedge \ldots \wedge v_{nm})\} = 0$$

Using De Morgan's law, we see that each conjunction of positive literals is logically equivalent to a disjunction of negative literals.

$$(\overline{(v_{11} \wedge v_{12} \wedge \ldots \wedge v_{1m})} = \overline{false}) \equiv ((\neg v_{11} \vee \neg v_{12} \vee \ldots \vee \neg v_{1m}) = true)$$

When represented in eCNF these disjunctions are identical to the clauses generated by the CNE algorithm.

$$((\neg v_{11} \vee \neg v_{12} \vee \ldots \vee \neg v_{1m}) = true) \equiv \{(\neg v_{11}), (\neg v_{12}), \ldots (\neg v_{1m})\} > 0$$

Therefore, the CNE constraint is logically equivalent to a CC constraint with @ =‘ =’ and $k = 0$.

Check Not Exist Count

The CNEC constraint uses double negation to succinctly express a parameterized set of CC constraints. This mimics a "for all" constraint without introducing new keywords or syntax to the language. The general pattern of the constraint is `CHECK NOT EXIST g(k @ (e))` where g is a query which contains the expression $(k@e)$ as a correlated subquery, k is a SQL integer expression, @ is an integer comparison from the set $\{>, \geq, <, \leq, \neq\}$, and e is the main query which returns a count of the result set. As k and e are both correlated to g, references to attributes from $\alpha(g)$ are replaced by values from the tuples returned by executing g.

As an example, consider the graph coloring problem augmented with a constraint forcing each color to be present a specific number of times via the relation *ColorNum(col, num)*. The corresponding SQL constraint would be

```
CHECK NOT EXIST(
  SELECT cn.col, cn.num FROM ColorNum WHERE (cn.num <>
    (SELECT COUNT(*) FROM Coloring cg1 WHERE cg1.col=cn.col)))
```

where g is the outer `SELECT` query, k is `cn.num`, @ = '$\neq$', and e is the inner `SELECT` query.

This constraint decomposes into one CC constraint for each color similar to the constraint described for the CC example above. The following algorithm converts a CNEC constraint into a set of CC constraints.

Algorithm *checkNEC(C)*

1. Input: C is a CNEC constraint, with top-level query g, main query e, and integer expression k.
2. $g' \leftarrow$ remove the expression (k @ e) from g.
3. $f \leftarrow$ a set of eCNF clauses, initially empty.
4. for each tuple $t \in g'()$
   - $k_g \leftarrow k\{t[A_1]/A_1, \ldots, t[A_n]/A_n\}$ where $A_i \in \mu(k|\alpha(g))$
   - $k'_g \leftarrow$ evaluate k_g
   - $e_g \leftarrow e\{t[A_1]/A_1, \ldots, t[A_n]/A_n\}$ where $A_i \in \mu(e|\alpha(g))$
   - $C' \leftarrow$ Create constraint "`CHECK` (k'_g @ e_g)"
   - $f = f \cup \{checkC(C')\}$
5. Return f

Figure 3.6: Algorithm for CNEC constraints

3.2.4 Translation Summary

In summary, we have described an RCSP specification language consisting of a subset of SQL's DDL conforming to a set of constraint patterns. We have provided algorithms for converting this input into a boolean satisfiability formula allowing us to take advantage of the efficiency of modern SAT solvers. In the rest of this chapter we discuss improvements that can be made to reduce overall solve time.

3.3 Problem Decomposition

A general notion in all problem solving is that of divide and conquer. It is often the case that large problems can be broken up into smaller and simpler problems whose solutions may be combined to provide a solution to the original problem. Although this technique is well studied in CSP [26], RCSP decomposition is, to our knowledge, an unexplored problem. One approach to reducing the size of an RCSP is to reduce the number of attributes in a constraint schema S, thereby reducing the number of possible S-tuples, and corresponding variables.

Definition 13. A *decomposition* of an RCSP $\Xi = (\mathcal{S}, \mathcal{C})$ is an RCSP $\Xi' = (\mathcal{S}', \mathcal{C}')$ such that for each $S \in \mathcal{S}$, there is a designated set $\psi(S) \subseteq \mathcal{S}'$ such that $S = \cup\psi(S)$.

Definition 14. A decomposition Ξ' of an RCSP Ξ is *valid* if for each solution $\mathbf{s}'$ of Ξ', there is a solution $\mathbf{s}$ of Ξ such that $r \in \mathbf{s}$ iff $\bowtie \mathbf{r}' = r$ where $\mathbf{r}' \subseteq \mathbf{s}'$ is the set of relations over the schemas of $\psi(\alpha(r))$ (the designated schemas of $\alpha(r)$).

The set of schemas $\psi(S)$ is a set of smaller relation schemas for replacing S. How the set is computed along with how constraints need to be modified to ensure validity determines the decomposition algorithm.

Proposition 3. *Suppose $\Xi = (\mathcal{S}, \mathcal{C})$ is an RCSP and A an attribute that appears in $\mathcal{S}$ but not referenced by any constraint $C \in \mathcal{C}$. Then the RCSP $\Xi' = (\mathcal{S}', \mathcal{C}')$ is a valid decomposition, where $\mathcal{S}'$ and $\mathcal{C}'$ are given as follows.*

1. $\mathcal{S}' = \{S' \mid S - \{A\}, S \in \mathcal{S}\} \cup \{R\}$, *where* R *is the singleton schema* $\{A\}$, *and*

$$\begin{aligned} \psi(S') &= \{S\} \textit{ if } A \notin S \\ &= \{S - \{A\}, R\} \textit{ otherwise} \end{aligned}$$

2. $\mathcal{C}'$ *are the constraints of* $\mathcal{C}$, *but with constraint schemas appropriately replaced by those in* $\mathcal{S}'$*:*

$$\begin{aligned} \{C' \mid\ & C' = C\{S'_1/S_1, ..., S'_k/S_k\} \textit{ for } C \in \mathcal{C}, \\ & \phi(C) = \{S_1, ..., S_k\}, S'_i = S_i - \{A\}, 1 \leq i \leq k\} \\ & \cup \{C_R\} \end{aligned}$$

where $C_R \equiv (\rho_{R_1}(R) \bowtie_{R_1.A \neq R_2.A} \rho_{R_2}(R)) = \emptyset$.

Proof. Note the constraint C_R is satisfied by any relation r of R that contains a single value from $\Delta(A)$.

Suppose $\mathbf{s}'$ is a solution of Ξ' and $r \in \mathbf{s}'$ is the relation for R. We define a solution $\mathbf{s}$ of Ξ. First, for each $s' \in \mathbf{s}'$, if $\alpha(s') = S$ for some $S \in \mathcal{S}$, then put s' in $\mathbf{s}$. If $\alpha(s') \cup \{A\} = S$ for some $S \in \mathcal{S}$, then put $s' \bowtie r$ in $\mathbf{s}$. Observe that $s' \bowtie r$ will have exactly the same number of tuples as s'. Clearly, there is a relation in $\mathbf{s}$ for each schema in $\mathcal{S}$.

Second, to show that $\mathbf{s}$ is a solution of Ξ, for each $C \in \mathcal{C}$, either $C = C'$ for some $C' \in \mathcal{C}'$ (i.e., no relation appearing in C contains A), or C contains a a schema $S \in \mathcal{S}$ that contains A. In the first case C trivially holds on $\mathbf{s}$ since $\mathbf{s}$ restricted to those relation schemas that do not contain A is identical to $\mathbf{s}'$. In the second case, since A is not referenced (does not participate in any condition of the constraint), C also

holds on $\mathbf{s}$. □

Remark. The proposition tells us that without loss of generality, we may assume that each attribute appearing in $\mathcal{S}$ is referenced by at least one constraint in $\mathcal{C}$.

Different strategies for computing $\psi(S)$ exist. We adopt an approach based on the following observation. A constraint that references only a subset of the attributes of a constraint relation schema can be more efficiently represented as a constraint over a smaller constraint relation schema consisting of only the referenced attributes.

Consider the RCSP Ξ_0:

$$(\{S_1(W_1, X_1, Y_1, Z_1), S_2(W_2, X_2, Y_2, Z_1)\}, \{C_1, C_2\})$$

over D. Suppose C_1 references attributes $S_1.W_1, S_1.X_1$ and $S_2.W_2, S_2.X_2$, and C_2 references attributes $S_1.Y_1, S_1.Z_1$ and $S_2.X_2, S_2.Y_2, S_2.Z_1$. Then a reasonable decomposition Ξ_1 is to use the set of constraint relations

$$\{S_3(W_1, X_1), S_4(Y_1, Z_1), S_5(W_2, X_2), S_6(X_2, Y_2, Z_1)\}$$

where $\psi(S_1) = \{S_3, S_4\}$, $\psi(S_2) = \{S_5, S_6\}$, and constraints $C_1' = C_1\{S_3/S_1, S_5/S_2\}$, $C_2' = C_2\{S_4/S_1, S_6/S_2\}$.

Remark. As in database normalization, additional constraints may be necessary to ensure the *lossless join decomposition* property, which is implied in the definition of

validity. As an example, if constraint C_1 above is a cardinality constraint that limits the number of occurrences of each value of $\Delta(X_1)$ to two, then a constraint must be placed on the join of S_1 and S_4 to ensure that C_1 will continue to hold over $s_3 \bowtie s_4$, for relations s_3 over S_3 and S_4 over R_4 of a solution of Ξ_1.

To address this we turn to the notion of key, more generally functional dependency. Functional dependencies represent some of the most common constraints in CSPs. An example is "a node can be assigned at most one color". A simple condition for ensuring lossless join decomposition is, for each $S \in \mathcal{S}$, to require that all relation schemas in $\psi(S)$ share a key of S. Combining this with our discussion above, we arrive at an approach for computing decomposition, which we call Constraint-Based decomposition (CB).

First, given an RCSP $\Xi = (\mathcal{S}, \mathcal{C})$, if $C \in \mathcal{C}$ and $S \in \phi(C)$, then $\mu(C|S)$, the set of attributes in S referenced by C, is a good candidate for establishing a new relation in the decomposition. If K_S is the set of attributes that form the *designated key* of S, then the *replacement schema* for S in C is $\mu^+(C|S) = \mu(C|S) \cup K_S$. The set of all replacement schemas of S is $\omega(S) = \{S' | S' = \mu^+(C|S), C \in \mathcal{C}\}$.

Definition 15. Let $\Xi = (\mathcal{S}, \mathcal{C})$, a *CB-decomposition* of Ξ is the decomposition $\Xi' = (\mathcal{S}', \mathcal{C}')$ given as follows.

1. $\mathcal{S}' = \cup_{S \in \mathcal{S}} \omega(S)$, and for each $S \in \mathcal{S}$, $\psi(S) = \omega(S)$.

2. $\mathcal{C}'$ are the constraints in $\mathcal{C}$ with constraint schemas replaced by their replace-

ment schemas:

$$\{C' | \quad C' = C\{S'_1/S_1, ..., S'_k/S_k\} \text{ for } C \in \mathcal{C}, \\ \phi(C) = \{S_1, ..., S_k\}, S'_i = \mu^+(C|S_i), 1 \leq i \leq k\} \\ \cup \beta$$

where β is the set of constraints that ensures solution consistency across elements of $\omega(S)$:

$$\beta \quad = \{C \mid \quad C \equiv \pi_Y(S'_1) = \pi_Y(S'_2), Y = S'_1 \cap S'_2, \\ \text{for } S'_1 \neq S'_2 \in \omega(S), S \in \mathcal{S}\}$$

Proposition 4. *CB-decomposition is valid.*

Proof. Given the original RCSP $\Xi = (\mathcal{S}, \mathcal{C})$, and the corresponding CB-decomposition $\Xi' = (\mathcal{S}', \mathcal{C}')$ with solution database $\mathbf{s}'$, we build the solution database $\mathbf{s}$ for Ξ.

1. For each constraint $C \in \mathcal{C}$, let C' be the corresponding constraint in Ξ', and $\phi(C')$ be the set of relation schemas in $\mathcal{S}'$ that appear C'. By definition, a set of solution relations h over $\phi(C')$ satisfies constraint C'. As $\mu(C|S) = \mu(C'|S)$ for all $S \in \mathcal{S}$, h also satisfies constraint C.

2. No relation in $\mathbf{s}'$ violates a constraint $C \in \mathcal{C}$. For each $s' \in \mathbf{s}'$

   - If $\mu(C|\alpha(s')) \subset \mu(C|S)$ for all $S \in \mathcal{S}$, then $\alpha(s')$ does not have contain a set of attributes constrained by C, and thus C is not violated.

   - If $\mu(C|\alpha(s')) = \mu(C|S)$ for a $S \in \mathcal{S}$, then $\alpha(s')$ is the $S' \in \mathcal{S}'$ used as a replacement for S in C' and holds by 1.

- If $\mu(C|\alpha(s')) \supset \mu(C|S)$ for a $S \in \mathcal{S}$, then s' is restricted by the β constraints in Definition 3.1 to the exact tuple combination obtained by the solution relations for C', where C' is the corresponding constraint for C in Ξ'.

3. By proposition 1, for each $S \in \mathcal{S}$ the union $\cup_{S' \in \omega(S)} S' = S$.

4. If r' is the set of relations over $\omega(S)$ for each $S \in \mathcal{S}$, then add $\bowtie r'$, the natural join of the relations in r', to $\mathbf{s}$.

   By 1 and 2, $\bowtie r'$ does not violate any constraint in $\mathcal{C}$.

   By 3, $\alpha(\bowtie r') = S$, therefore all $s \in \mathbf{s}$ are constructed.

In summary, as CB-decomposition is a generic approach for any finite RCSP with at least one constraint schema containing 3 or more attributes. Although it does not guarantee performance improvement, empirical results in Section 6.x demonstrate important potential speed up. Finally, the algorithm is automatable via SQL parse tree analysis and manipulation, and thus highly useful for SQL CSP solvers.

3.4 Solver Selection

There are many approaches to solving finite CSP. Three common ones are backtrack based propositional logic inference, local search, and combinatorial search. Although there are many different algorithms for these approaches, few support the conjunction

of variables inside constraints. Hence, eCNF requires an important simplification step before CSP algorithms can be applied.

The eCNF simplification algorithm removes all conjunctions inside a clause. The algorithm is given in Figure 3.7. Note that the @ = '=' case is the most expensive

Algorithm *simplify*(c)

1. Input: $c = \mathcal{F}@k$ is an eCNF clause, where $\mathcal{F} = \{F_1, \ldots, F_n\}$, and F_i is a conjunction of boolean literals $(l_1^i \wedge \ldots \wedge l_m^i)$.
2. for each $F_i \in \mathcal{F}$ where $|F_i| > 1$
   - introduce a new boolean variable u_i,
   - replace the F_i in $\mathcal{F}$ with u_i.
   - if $@ \in \{>, \geq, =\}$, impose constraint $u_i \Rightarrow (l_1^i \wedge \ldots \wedge l_m^i)$ with eCNF clauses: $\{(\neg u_i), (l_j^i)\} > 0$ for $1 \leq j \leq m$
   - if $@ \in \{<, \leq, =\}$, impose constraint $u_i \Leftarrow (l_1^i \wedge \ldots \wedge l_m^i)$ with eCNF clause: $\{(u_i), (\neg l_1^i), \ldots, (\neg l_m^i)\} > 0$

Figure 3.7: Algorithm for eCNF simplification

as the implication must be imposed in both directions. The idea, known as variable substitution, is to introduce a new variable for each conjunction.

Proposition 5. *Formulas c and simplify(c) are logically equivalent modulo the substitution variables.*

Proof. Let L represent all solutions to an original eCNF clause c, and L' represent all solutions to *simplify*(c). For each solution in L there exists at least one solution in L' with identical assignments for all the original variables. Such a solution in L' can

be found by extending the original solution with new assignments for each variable u_i such that $u_i = (l_1^i \wedge \ldots \wedge l_m^i)$ where $(l_1^i \wedge \ldots \wedge l_m^i)$ is the original conjunction u_i replaced in $\mathcal{F}$. Similarly, each solution in L' has a corresponding solution in L retrieved by simply removing all substitution variables u_i.

With the simplification algorithm, eCNF translates easily to input forms commonly used for several major algorithms. Examples are the Davis Putnam Logemann Loveland (DPLL) algorithm for propositional clauses, the WalkSAT algorithm for randomized local search, and the Branch and Bound algorithm for Integer Linear Programming. The ability to leverage a variety of solvers is key to providing efficient solve times for a wide variety of problems.

Chapter 4

Implementation Details

In this chapter we describe the *SQL Constraint Data Engine* (SCDE) prototype for testing general concepts and techniques related to RCSP processing. We focus on concrete engineering design decisions made in order to realize the concepts described in Chapter 3.

The SCDE is a system designed as a combination of modules enabling flexible modification and clean separation of functionality. Figure 4.1 shows the architecture of the system. Like traditional RDB systems, SCDE has an Application Program Interface (API) that allows for a variety of user interfaces. Although at this time we have only implemented a file interface for processing SQL directly, other interfaces such as an interactive command line, or graphical query tool could be easily implemented. The SCDE kernel, shaded in grey, is responsible for all the RCSP related work. It receives specifications through the API in the SCDE Command Language (SCL), which is a slight extension to SQL's DDL. These specifications are parsed and built into problem models which can then be optimized and converted into boolean

SAT problems in eCNF. Finally, the eCNF is then converted to a solver specific format, and an external solver is used to solve the CSP. As oppose to writing our own relational database implementation, SCDE use SQLite, an "in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.",[1] The rest of this chapter describes in detail the SCL, the RCSP processor, the eCNF solver, and the various backend SAT solvers used.

4.1 The SCDE Command Language

Mostly overlooked in other CSP tools but central to the design of SCDE's Command Language (SCL) is enabling intuitive and frequent user interactions with metadata objects. Frequent user interaction is a basic premise of dynamic CSPs. Furthermore, there is growing recognition that the ability to incrementally fine-tune constraints is important to fully exploit the power of intelligent tools in solving real-world problems [32]. Table 4.1 shows the basic SQL extensions to support the specification of RCSPs in SCDE. This syntax is aimed at flexible user interaction and aligns well with the standard SQL model for administering metadata objects. Although Section 3.1 demonstrates that no extension to SQL is strictly necessary, we choose to make a few extensions to distinguish RCSP commands from traditional SQL commands.

[1] http://www.sqlite.org/; Sqlite has been adopted as the back-end of major software tools including Adobe Photoshop Lightroom, Mac OS-X, and the Firefox Web-Browser.

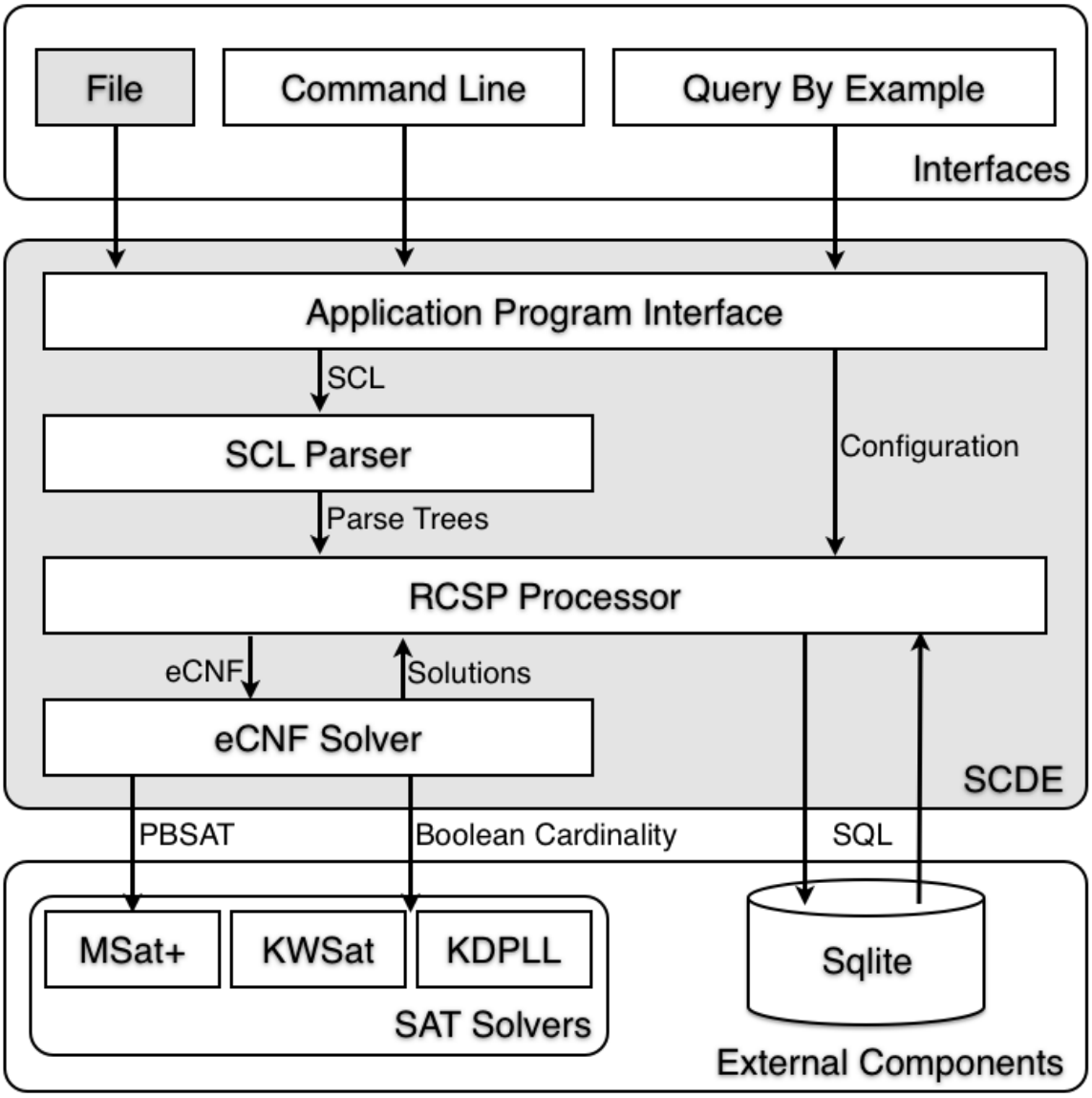

Figure 4.1: SCDE Architecture

1	CREATE CONSTRAINT TABLE <Q> ({<col> <type> FOREIGN KEY REFERENCES <ftable>(<fcol>),} [KEY (<col>[,<col>]),] {CONSTRAINT <c> <constraint>,}
2	ALTER CONSTRAINT TABLE <Q> (ADD CONSTRAINT <c> <constraint>);
3	ALTER CONSTRAINT TABLE <Q> (DROP CONSTRAINT <c>);
4	TRACE CONSTRAINT TABLE <Q> CONSTRAINT <c>;
5	SOLVE TABLE <Q>;
6	COMMIT SOLUTION;
7	DROP CONSTRAINT TABLE <Q>;

Table 4.1: SCDE Command Language

The `CREATE CONSTRAINT TABLE` command declares the associated table to be a constraint relation (see Section 1.5) or *constraint table*. Each column (or attribute) of the constraint table must be a foreign key, i.e. a key belonging to a different table. We refer to this set of foreign tables as *base tables*. Note that the domain of any attribute in a constraint table is simply the projection of the corresponding base table onto the foreign key, and thus easy to calculate. Base tables comprise the database D in the RCSP definition given by Section 1.5, and can be either regular tables or previously solved constraint tables.

Constraints can be specified in two different ways. First, one key may be included as part of the specification. This key is the *designated key* used by the decomposition algorithm in Section 3.3. Second, a set of constraints adhering to one of the patterns described in Section 3.2.3 may be included as part of the specification via the

`CONSTRAINT <c>` syntax where `<c>` is an arbitrary name assigned to the constraint.

Although the `CREATE CONSTRAINT TABLE` command is sufficient for specifying RCSP, other commands enable important features for user interaction with the problem. The `ALTER TABLE` statement allows these constraints to be added or removed in subsequent commands. The `TRACE` command reports constraint metrics to the user described Sections 5.2 and 5.3. These allow a user to get some feedback on constraints before initiating the solving process. The `SOLVE` command triggers the compiling, solving, and populating of the constraint table. Finally, the `COMMIT SOLUTION` command writes the current results (of all active constraint tables) to disk.

4.2 The RCSP Processor

The RCSP Processor is responsible for extracting RCSPs from parse trees generated by the SCL Parser and creating logically equivalent boolean SAT problems in eCNF. It's functionality is divided into the four modules shown in Figure 4.2.

4.2.1 Problem Modeler

The Problem Modeler organizes the parse tree into a model which facilitates analysis of attributes, constraint tables, and constraints. This process organizes the constraints according to their pattern types, and creates the constraint relations where the solution, once found, will be stored. Depending on configuration pa-

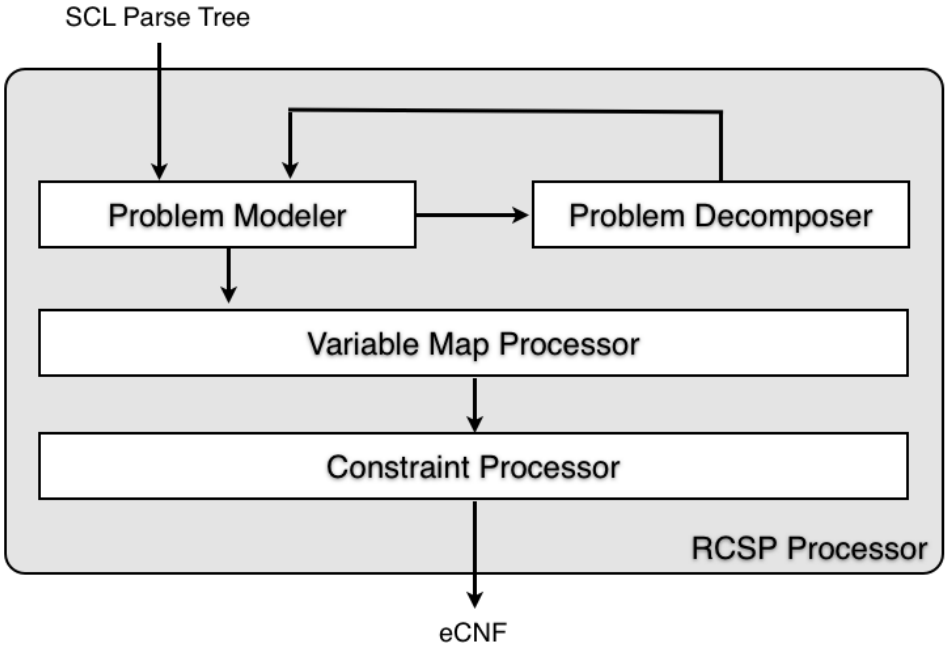

Figure 4.2: The RCSP Processor

rameters, the model is either sent to the the Problem Decomposer to undergo the CB-Decomposition process described in Section 3.3, or to the Variable Map Processor.

4.2.2 Variable Map Processor

The Variable Map Processor is responsible for creating, populating and filtering variable maps. For each constraint table schema $S \in \mathcal{S}$, a variable map is created with the schema $\{vbid \cup S\}$. Furthermore, each variable map is assigned a unique id allowing variables from different maps to be distinguished.

A variable map is initially populated with $\Delta(S)$ and then filtered by *filter con-*

straints. A filter constraint is a CNE constraint which contains a single constraint schema reference to S. As the CNE algorithm dictates, this results in a set of eCNF clauses which force selected variables to $false$. Rather than forwarding these clauses to the solver, it is much more efficient to simply remove these variables from the corresponding map. This is accomplished by the algorithm in Figure 4.3. As detailed in Section 5.2, the algorithm reduces the total number of variables used to process the rest of the RCSP. Therefore, SCDE processes all filter constraints before any others.

Algorithm *filter*(C)

1. Input: C is a CNE constraint with $\phi(C) = \{S\}$,
   e is the main query of C
2. $e' \leftarrow vb(e)$
3. Delete all tuples in $e'()$ from $\nu(S)$

Figure 4.3: Algorithm for filter constraints

4.2.3 Constraint Processor

The Constraint Processor is responsible for processing two genres of constraints. The first consists of `CHECK` constraints adhering to one of the four patterns CE, CNE, CC, or CNEC. The algorithms for processing these constraints are detailed in Section 3.2.3. The second genere consists of functional dependency constraints specified as *keys*.

Key constraints play a very important role in decomposition. They are enforced by cardinality constraints that require, for all variables mapped to the same key value, at most one is assigned to *true*. As an example, consider the course scheduling example in Section 3. Assuming {cid}is the key in the constraint schema schedule(cid, pid, tid), if there are two professors $\Delta(pid) = \{$'bob', 'lisa'$\}$ and two time slots $\Delta(tid) = \{$'10am', '11am'$\}$, then for each course, there are four possible tuples, and therefore four variables. If $v_1 = \{$'cs1', 'bob', '10am'$\}$, $v_2 = \{$'cs1', 'bob', '11am'$\}$, $v_3 = \{$'cs1', 'lisa', '10am'$\}$, $v_4 = \{$'cs1', 'lisa', '11am'$\}$ then the eCNF clause $\{v_1, v_2, v_3, v_4\} \leq 1$ enforces the key constraint for the 'cs1' value in $\Delta(cid)$.

We use the notation $uni(S, K)$ to represent the set of eCNF clauses enforcing the uniqueness constraint $t_1[K] \neq t_2[K]$ for all distinct tuple pairs t_1, t_2 of a relation over the constraint schema S with key K.

Algorithm $uni(S, K)$

1. Input: S is a constraint schema with key $K = \{k_1, \ldots, k_n\}$ and $\Delta(K) = \Delta(k_1) \times \ldots \times \Delta(k_n)$.
2. for each tuple $t = \{x_1, \ldots, x_n\} \in \Delta(K)$:
   - create the eCNF clause: $\{\pi_{vbid}(\sigma_{K=t}\nu(S))\} \leq 1$

   where $K = t$ is is short for $k_1 = x_1 \wedge \ldots \wedge k_n = x_n$

Figure 4.4: Algorithm for unique key enforcement

If several constraint schemas share the same key, as is the case for replacement

schemas in CB-decomposition, then the Constraint Processor enforces that the key values chosen across all such schemas are identical. For example, assume constraint schemas in set $\mathcal{H} = \{H_1, \ldots, H_n\}$ all share the same key K. The algorithm requires that if a key value occurs in one relation over schema H in $\mathcal{H}$, then that same key value must be present in all other relations over schemas in $\mathcal{H}$. Since key values are unique in each relation, this effectively enforces that key values across all relations over schemas in $\mathcal{H}$ are identical. The algorithm for enforcing this constraint with eCNF clauses is given in Figure 4.5.

Algorithm *equi*$(\mathcal{H}, K)$

1. Input: $\mathcal{H}$ a set of constraint schemas sharing key $K = \{k_1, \ldots, k_n\}$ and $\Delta(K) = \Delta(k_1) \times \ldots \times \Delta(k_n)$.
2. $H' \leftarrow$ a schema in $\mathcal{H}$ with the minimum number of attributes.
3. for each $H \in \mathcal{H}, H \neq H'$:
   - for each tuple $t = \{x_1, \ldots, x_n\} \in \Delta(K)$:
     - impose the constraint: $\vee\{\pi_{vbid}(\sigma_{K=t}\nu(H'))\} \Leftrightarrow \vee\{\pi_{vbid}(\sigma_{K=t}\nu(H))\}$
       - for each $vbid \in \{\pi_{vbid}(\sigma_{K=t}\nu(H'))\}$
         - create eCNF clause: $\{\neg vbid\} \cup \{\pi_{vbid}(\sigma_{K=t}\nu(H))\} > 0$
       - for each $vbid \in \{\pi_{vbid}(\sigma_{K=t}\nu(H))\}$
         - create eCNF clause: $\{\neg vbid\} \cup \{\pi_{vbid}(\sigma_{K=t}\nu(H'))\} > 0$

     where $K = t$ is is short for $k_1 = x_1 \wedge \ldots \wedge k_n = x_n$

Figure 4.5: Algorithm for key tuple equality amongst multiple relations

All constraint algorithms implemented by the constraint processor generate eCNF

clauses that are combined, via conjunction, into one eCNF sentence. As there can be multiple constraint schemas referenced in a single constraint, every literal is composed of a VBmap id and a (possibly negated) variable id. Once generated, eCNF clauses are sent to the *eCNF Solver*.

4.2.4 Problem Decomposer

The Problem Decomposer implements the CB-Decomposition defined in Section 3.3. It receives a problem model corresponding to an RCSP $\Xi = (\mathcal{S}, \mathcal{C})$, and returns a decomposed model representing a new RCSP $\Xi' = (\mathcal{S}', \mathcal{C}')$. We incorporate certain design/implementation choices to improve efficiency. The current implementation assumes a single constraint schema, S_0, in the input RCSP. We denote by K_{S_0} the designated key of S_0.

The set $\mathcal{S}'$ is derived from dividing S_0 according to Definition 15.1. To ensure a reduction in the size of the problem representation, we add the restriction that all resulting schemas of $\mathcal{S}'$ are strict subsets of S_0. This avoids the generation of the same VBmap in Ξ' as in Ξ which, in practice, significantly reduces the total number of boolean variables represented in the eCNF. The tradeoff for this design choice is that *full constraints*, which are constraints that references all attributes of S_0 (i.e., $\mu^+(C|S_0) = S_0$), need special handling since S_0 will not be present in $\mathcal{S}'$. Furthermore, since a decomposition must satisfy $\cup\mathcal{S}' = S_0$, problems that have

full constraints may need the addition of a pseudo constraint schema to ensure that solutions to S_0 may be recovered from solutions of $\mathcal{S}'$.

In order to compute the solution to Ξ from the solution to Ξ' and to translate full constraints to eCNF, a set of constraint schemas which covers all the attributes of S_0 must be joined. Although $\cup\mathcal{S}' = S_0$, any subset $J \subseteq \mathcal{S}'$ for which $\cup J = S_0$ is sufficient. By selecting J to be the minimum set cover of $\mathcal{S}'$ over S_0, we reduce the time of the required join operations. This is especially relevant for full constraints, as the constraint translation algorithms use VBmaps in place of constraint schemas, and the join of the VBmaps of J can be significantly less expensive than the join of the VBMaps of $\mathcal{S}'$. Note that when multiple minimum set covers exist, we prioritize schemas in $\mathcal{S}'$ with the fewest attributes. This leads us to our algorithm for CB-decomposition shown in Figure 4.6.

The algorithm references two other constraints, *uni* and *equi* which are detailed in Section 4.2.3. When combined, these two constraints serve as a simplified version of the β constraints described in Definition 15.2, enabling the key-based join of the relations over $\mathcal{S}'$ to form a solution to the original RCSP. Unfortunately, *uni* and *equi* do not form a complete implementation of the β constraints because a distinct pair of constraint schemas S'_i, S'_j, may have an overlapping set of attributes $S'_i \cap S'_j$ that contains more attributes than just the designated key. As a result, all possi-

Algorithm *decomp*(Ξ)

1. Input: an RCSP $\Xi = (\{S_0\}, \mathcal{C})$ with designated key K_{S_0}
2. initialize $\mathcal{S}' = \emptyset$ and $\mathcal{C}' = \emptyset$
3. for each $C \in \mathcal{C}$ that is not a full constraint (i.e. $\mu^+(C|S_0) \subset S_0$) :
   - $\mathcal{S}' \leftarrow \{\mu^+(C|S_0)\} \cup \mathcal{S}'$
   - $\mathcal{C}' \leftarrow \{C\{S'/S_0\}\} \cup \mathcal{C}'$
   - $\mathcal{C}' \leftarrow \{uni(S', K_{S_0})\} \cup \mathcal{C}'$
4. $J \leftarrow$ minimal subset of $\mathcal{S}'$ covering S_0
5. for each $C \in \mathcal{C}$ that is a full constraint (i.e. $\mu^+(C|S_0) = S_0$) :
   - $\mathcal{C}' \leftarrow \{C\{(\bowtie J)/S_0\}\} \cup \mathcal{C}'$
6. $\mathcal{C}' \leftarrow \{equi(\mathcal{S}', K_{S_0})\} \cup \mathcal{C}'$

Figure 4.6: Algorithm for CB-Decomposition

ble decompositions are not covered by our algorithm. Fortunately, for most RCSP, choosing the best designated key typically circumvents the need for the computationally expensive β constraints.

4.3 The eCNF Solver

The eCNF solver is responsible for solving boolean satisfiability problems in eCNF form. It is composed of the three modules shown in Figure 4.7. The purpose of each module is to reduce the eCNF into a form more closely aligned with a back-end solver's expected input.

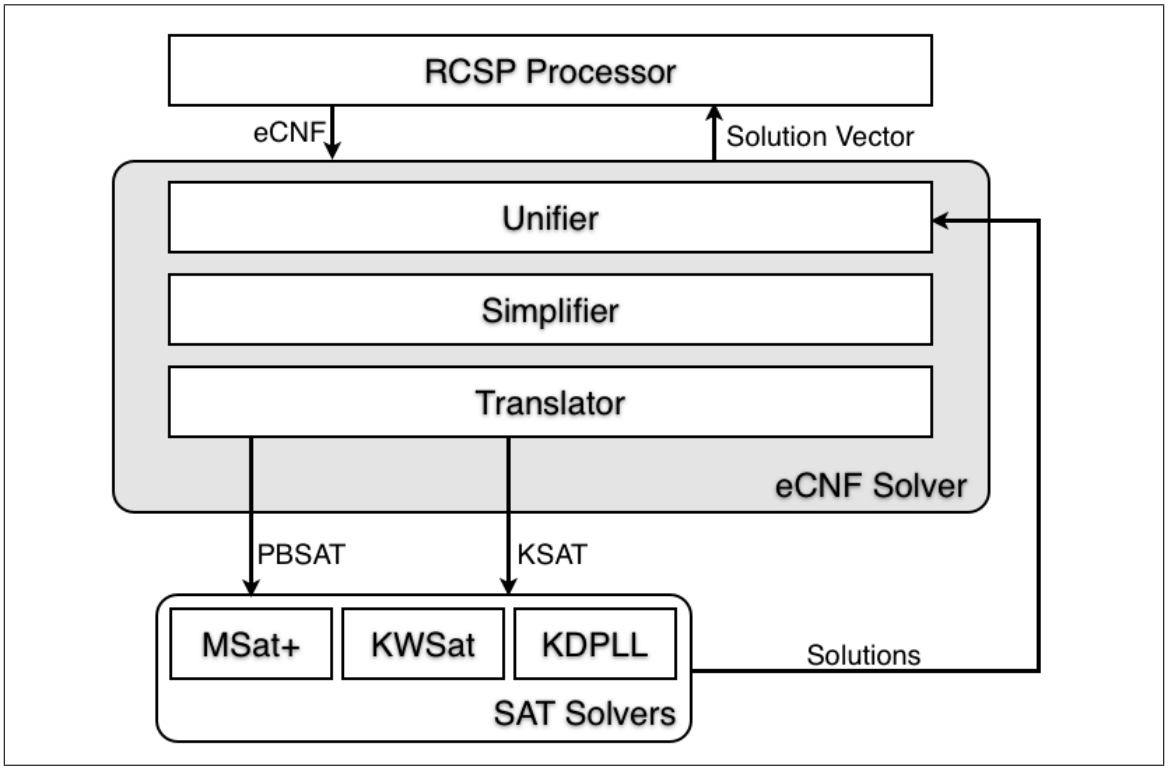

Figure 4.7: The eCNF Solver

4.3.1 Unifier

Solvers typically require the set of all variable ids to be a contiguous set of integers. The *Unifier* maps the composited ids of eCNF, each consisting of a VBMap id and variable id, into a contiguous set of integers. This mapping is done via a hash table, called the global map, that correctly compensates for positive and negative literals. The global map is also used to translate a computed solution back to composite ids, allowing the SCDE system to populate the constraint relations with tuples from VBMaps that correspond to the solution.

4.3.2 Simplifer

The *Simplifier* module implements the variable substitution algorithm discussed in Section 3.7. The new variables that are created throughout this process serve only to impose restrictions on original variables. Although these variables are entered into the global map, any assignments made by the solver to these variables are ignored.

4.3.3 Translator

Once unified and simplified, an eCNF sentence is a conjunction of clauses $\mathcal{F}$ @ k where $\mathcal{F}$ is a set of literals, @ is a comparison operator from the set $\{>, \geq, <, \leq, =\}$, and k is an integer. The *Translator* module is responsible for making the final conversion to a solver input format. Currently, it supports two formats, *Boolean*

Cardinality (BC), and *Pseudo Boolean* (PB).

The BC format for each clause is $\sum_i l_i \geq k$ for $1 \leq i \leq n$, where each l_i is a literal, and k is an integer > 0. The algorithm in Figure 4.8 implements the translation. We note that if the resulting BC clause has $k \leq 0$, then it is trivially false.

Algorithm $toBC(c)$

1. Input: c is a simplified eCNF clause of the form $\mathcal{F}$ @ k
   $\mathcal{F} = \{F_1, \ldots, F_n\}$
2. if @ = '=', then return $toBC(\mathcal{F} \leq k) \wedge toBC(\mathcal{F} \geq k)$
3. if @ = '<', then $k = k - 1$ and @ = '$\leq$'
4. if @ = '>', then $k = k + 1$ and @ = '$\geq$'
5. if @ = '$\leq$', then $k = n - k$, @ = '$\geq$', and $F_i = \neg F_i$ for all $F_i \in \mathcal{F}$
6. return BC clause: $\sum_{1 \leq i \leq n} F_i \geq k$

Figure 4.8: Algorithm for translating eCNF to BC

The PB format is $\sum_i w_i * v_i \geq k$ for $1 \leq i \leq n$, where w_i is an integer weight, and v_i is a boolean variable. The algorithm for the translation is given in Figure 4.9. Note that there is no concept of negated variables in this format. Instead, we replace $\neg v$ with $1 - v$. As a result, k can be meaningful when zero or negative. For example, consider the eCNF expression $\{v_1, \neg v_2, \neg v_3\} \geq 1$. When we apply the $toPB$ algorithm we arrive at the expression $1 * v_1 - 1 * v_2 - 1 * v_3 \geq -1$.

Algorithm *toPB*(c)

1. Input: c is a simplified eCNF clause of the form $\mathcal{F}$ @ k
   $\mathcal{F} = \{F_1, \ldots, F_n\}$
2. if @ = '=', then return $toPB(\mathcal{F} \leq k) \wedge toPB(\mathcal{F} \geq k)$
3. for each $F_i \in \mathcal{F}$:
   - if F_i is a negated literal: $w_i = -1$ and $k = k - 1$
   - else: $w_i = 1$
   - v_i is the variable id in F_i
4. if @ = '<', then $k = k - 1$ and @ = '$\leq$'
5. if @ = '>', then $k = k + 1$ and @ = '$\geq$'
6. if @ = '$\leq$', then $k = -k$, @ = '$\geq$', and $w_i = -1 * w_i$ for $1 \leq i \leq n$
7. return PB clause: $\sum_{1 \leq i \leq n} w_i * v_i \geq k$

Figure 4.9: Algorithm for translating eCNF to PB

4.3.4 Back-End Solvers

The SCDE system currently supports 3 back-end solvers. The first solver, *MiniSat+* [13], is a Pseudo Boolean solver that uses a variety of algorithms to translate PB clauses into CNF clauses. MiniSat[12] is a state of the art, DPLL based, SAT solver which incorporates many modern heuristics including watched literals, conflict driven back tracking, and dynamic variable ordering.

The other two solvers are our own implementations of existing algorithms. The *KWSAT* solver is an implementation of the WalkSAT algorithm[14] adapted for

the BC format. The *KDPLL* solver is an implementation of the DPLL algorithm adapted for the BC format based on [11]. Together these algorithms allow us to gain perspectives on SCDE's compilation strategy and test the relative strengths of local vs complete search algorithms for RCSPs.

Chapter 5

Theoretical Analysis

This chapter shows some theoretical properties of SCDE and the algorithms we have developed. We begin by demonstrating that our supported syntax allows us to express NP-complete problems. We then give some size metrics on our generated boolean satisfiability problems. These metrics can help predict the amount of time required to solve certain problems and give insight on what types of SQL constraints to avoid in the problem specification process.

5.1 On the Expressiveness of SCL

In this section we demonstrate how to express any 3-SAT problem in the SCL. The purpose of this is to verify that our subset of SQL DDL is indeed capable of expressing NP-complete problems, and to highlight some of the advantages of our count based SQL constraint patterns.

```
CREATE TABLE variables(id);
    INSERT INTO variables VALUES(1);
    INSERT INTO variables VALUES(2);
    INSERT INTO variables VALUES(3);
    INSERT INTO variables VALUES(4);

CREATE TABLE boolean(val, name);
    INSERT INTO boolean VALUES(0, 'false');
    INSERT INTO boolean VALUES(1, 'true');
```

Figure 5.1: SQL for 3-SAT base tables.

5.1.1 3-SAT

The 3-SAT problem is a subset of the boolean satisfiability problem which is limited to a conjunction of clauses where each clause consists of a disjunction of 3 literals. For example, consider a problem with 4 boolean variables: $v_1..v_4$ and the implication constraint $v_1 \wedge v_2 \implies v_3 \wedge \neg v_4$. The problem can be expressed with the 3-SAT sentence $(\neg v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_4)$. An example satisfying assignment is $v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 0$.

Before we can specify the 3-SAT problem above in SQL, we need to create and populate tables representing the list of variables and domain. This can be accomplished with the SQL statements in figure 5.1. Given this set of base tables, we can specify the problem, given in Figure 5.2, by creating a constraint schema which maps the variables.id attribute to the boolean.val attribute and adding an integrity constraint for each 3-SAT clause.

Constraints $C1, C2$ use the CE constraint type to express the two clauses of the

```
CREATE CONSTRAINT TABLE sol(
        vid INTEGER FOREIGN KEY REFERENCES variables(id),
        asg INTEGER FOREIGN KEY REFERENCES boolean(val),

        CONSTRAINT C1 CHECK (EXISTS
                (SELECT * FROM sol s WHERE
                (s.vid=1 AND s.asg=0) OR
                (s.vid=2 AND s.asg=0) OR
                (s.vid=3 AND s.asg=1))),

        CONSTRAINT C2 CHECK (EXISTS
                (SELECT * FROM sol s WHERE
                (s.vid=1 AND s.asg=0) OR
                (s.vid=2 AND s.asg=0) OR
                (s.vid=4 AND s.asg=0))),

        CONSTRAINT C3 CHECK (NOT EXISTS
                (SELECT * FROM sol s1, sol s2 WHERE
                s1.vid=s2.vid AND s1.asg<>s2.asg)));
```

Figure 5.2: SQL specification for 3-SAT problem.

3SAT sentence. The final constraint ($C3$) uses the CNE constraint type to enforce that no variable be given two values. This constraint enforces a traditional notion of variables that is implicitly understood by typical CSP solvers. As a typical RDB has no notion of CSP variables, it is sometimes necessary to explicitly state such constraints. Note that this rule could be replaced with the declaration that *vid* is a key.

5.1.2 Count Constraints

Although any NP-complete problem can be encoded in 3-SAT some simple constraints are prohibitively large to encode manually. One case is the cardinality constraint, where a specific number of variables are required to be true. For example, given the boolean variables $v_1..v_4$, the following 3-SAT sentence (set of clauses) is

required to ensure that exactly 2 variables are set to *true*.

$(v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee v_4) \wedge (v_1 \vee v_3 \vee v_4) \wedge (v_2 \vee v_3 \vee v_4) \wedge$

$(\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_4) \wedge (\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (\neg v_2 \vee \neg v_3 \vee \neg v_4)$

In this section we describe how the count constraint patterns CC and CNEC patterns can succinctly encode certain problems. For example, the CC pattern allows us to express cardinality constraints like the one above with the following syntax.

```
CONSTRAINT C1 CHECK
        (2 =  (SELECT COUNT(*) FROM sol s WHERE s.asg=1))
```

One of the disadvantages of our 3-SAT encoding so far is the need to express each 3SAT clause as a separate SQL constraint. To eliminate this requirement, we can create a new base table that stores the data representing all of the clauses (see figure 5.3). This table can then be used in conjunction with the CNEC constraint pattern to create one SQL constraint which enforces all 3-SAT clauses in the problem. The CNEC constraint, given in Figure 5.4, essentially creates a set of CC constraints from the subquery with parameters passed in from the column result set of the parent query. Note the double negative nature of this pattern using the NOT to negate the comparison '>', resulting in CC constraints of the form `CHECK (1 <= (SELECT COUNT(*) ...))` which are identical (see Section 3.2.3) to the original CE constraints in Figure 5.2.

```
CREATE TABLE clauses(id, vid1, asg1, vid2, asg2, vid3, asg3);
INSERT INTO clauses VALUES(1, 1, 0, 2, 0, 3, 1);
INSERT INTO clauses VALUES(2, 1, 0, 2, 0, 4, 0);
```

Figure 5.3: SQL for base table describing $(\neg v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_4)$.

```
CONSTRAINT C1 CHECK (NOT EXISTS
        (SELECT c.vid1, c.asg1, c.vid2, c.asg2, c.vid3, c.asg3
                FROM clauses c WHERE
                1 > (SELECT COUNT(*) FROM sol s WHERE
                        (s.vid=c.vid1 AND s.asg=c.asg1) OR
                        (s.vid=c.vid2 AND s.asg=c.asg2) OR
                        (s.vid=c.vid3 AND s.asg=c.asg3))))
```

Figure 5.4: The CNEC SQL constraint for 3-SAT.

In summary, the CNEC constraint is our most expressive constraint pattern. It's syntax is sufficient for expressing many NP-complete problems succinctly and it is used widely in our SQL CSP specifications.

5.2 Variable Maps

In this section we analyze the size of variable maps, and the total number of boolean variables generated for a problem. This is impacted by the size of each attribute's domain, $|\Delta(A)|$, filter constraints, and problem decomposition.

Given a constraint schema $S = \{A_1 \dots A_n\}$, the corresponding variable map, $\nu(S)$, contains every possible S-tuple, along with a unique identifier for each tuple representing the boolean variable id. Thus the total number of variables generated for the constraint relation S is $vbsize(S) = |\nu(S)| = \prod_{1 \leq i \leq n} |\Delta(A_i)|$. This can be further reduced by filter constraints (see Section 4.2.2).

Given a single filter constraint FC with $\phi(FC) = S$ and main query e, the size of the filter is $|FC| = |e()|$. Recall $e()$ is the set of tuples returned by e. Each filter reduces $|\nu(S)|$ by $|FC|$. Thus, given a set of filter constraints $FC_1 \ldots FC_m$ with $\phi(FC_j) = S$, the $vbsize(S) = |\nu(S)| = \prod_{1 \leq i \leq n} |\Delta(A_i)| - \sum_{1 \leq j \leq m} |FC_j|$.

For some situations, filter constraints can have a powerful impact on total variable size. In many situations, however, the total number of variables is dominated by the domain size of attributes, and the large cross product resulting from constraint schemas with many attributes. This can be mitigated by the CB-Decomposition algorithm.

Given a constraint schema $S = \{A_1 \ldots A_n\}$, the CB-Decomposition algorithm produces a set of constraint schemas $\psi(S) = \{S'_1, \ldots, S'_p\}$, where each $S'_i \subset S$. If $\mathcal{B}_i = \{B_{i1}, \ldots, B_{iq}\}$ is the set of attributes in S not present in S'_i, then $|\nu(S'_i)| = \frac{|\nu(S)|}{\prod_{1 \leq j \leq q} |\Delta(B_{ij})|}$ and the $vbsize(S) = \sum_{1 \leq i \leq p} \frac{\nu(S)}{\prod_{1 \leq j \leq q} |\Delta(B_{ij})|}$. As we will see in the experimental results section, this can lead to a substantial reduction in the total number of boolean variables.

In summary, an RCSP with $\mathcal{S} = \{S_1 \ldots S_n\}$ has a total number of variables given by $vbsize(\mathcal{S}) = \sum_{1 \leq i \leq n} |vbsize(S_i)|$. The total number of variables plays a role in determining the size of the search space, $2^{vbsize(\mathcal{S})}$, and provides an upper bound on the worst case solving time. For the many situations, however, the number and type

of boolean clauses are a better predictor of solving time.

5.3 Boolean Clauses

The size and type of boolean clauses generated by a constraint C depends on the constraint pattern used, the number of constraint schemas referenced ($|\phi(C)|$), and the number of tuples returned by the main query ($|e()|$).

5.3.1 Check Exists

As the algorithm in Figure 3.3 describes, a CE constraint generates a single eCNF clause with $|e()|$ conjunctions. Each conjunction has one literal from each $S \in \phi(C)$. If $|\phi(C)| = 1$, then there is only one literal in place of each conjunction, and the final encoding is a eCNF clause consisting of a disjunction of $|e()|$ literals. If $|\phi(C)| > 1$ then the *simplify* algorithm described in Section 3.4 is used to replace each conjunction by a substitution variable and $|\phi(C)|$ additional clauses of size 2. The total number of clauses is therefore $1 + (|\phi(C)| \times |e()|)$, where the first clause has $|e()|$ literals, and the rest have 2.

5.3.2 Check Count

As stated in Section 3.2.3, the CC constraint is processed similarly to a CE constraint. There are two important differences. First, the comparison can be one of $\{>, \geq, <$

$, \leq, =\}$. In the most expensive case, '=', the variable substitution scheme requires $(|\phi(C)| \times |e()|)$ clauses of size 2, and $|e()|$ extra clauses of size $(1+|\phi(C)|)$. This results in a total number of clauses equal to $1 + |e()| + (|\phi(C)| \times |e()|)$. The other difference is that the constraint can have high cardinality. Although eCNF can capture this in one clause, the same encoding in CNF can be very expensive. For example, consider the eCNF clause $\{(v_1) \ldots (v_n)\} = k$. In CNF, this expands to $\binom{n}{k-1}$ disjunctions containing $n-k+1$ positive literals, and $\binom{n}{k+1}$ disjunctions containing $k+1$ negative literals for a total of $\frac{n!}{(n-k+1)!(k-1)!} + \frac{n!}{(n-k-1)!(k+1)!}$ clauses. Although we choose SAT solvers capable of handling cardinality constraints, for some solvers the size of k can have a large impact on the solve time.

5.3.3 Check Not Exist

The CNE constraint is the simplest in terms of clause size, and translation to CNF. It produces $|e()|$ CNF clauses of size $|\phi(C)|$.

5.3.4 Check Not Exist Count

The CNEC constraint produces a set of CC constraints, each in turn producing eCNF clauses. The total number of CC constraints produces is $|e()|$, the number of tuples returned by the query g in the pattern `CHECK NOT EXIST g (k @ (e))`.

Chapter 6

Experiments and Results

This chapter analyzes the performance of SCDE on several CSP problems. Our goal in this analysis is not to demonstrate a faster approach to solving CSPs over existing CSP solvers, but to validate the feasibility of our approach and SQL CSP systems in general. In order to make comparisons to other traditional solvers, we separate SCDE's run time into translation time, solve time, and total time. The translation time encompasses the SQL query and file processing necessary to create the boolean encoding fed to the backend SAT solver. The solve time is the amount of time the backend SAT solver takes to find a satisfying solution or discover that the problem is infeasible. The total time, or run time is the the sum of the translation and solve times. Notice that the time to reverse translate the boolean solution into a set of S-tuples and populate the s-relations is not accounted for. This time is almost always insignificant as most solutions are very small in comparison to their search spaces.

The first solver used for comparison is the IBM ILOG CPLEX Optimizer. It is an industry standard tool for solving combinatorial problems and provides a wide variety of functionality. Although CPLEX supports multiple input formats, we restrict our experiments to the Optimization Programming Language (OPL) [15], and CPLEX's OPL solver.

The second solver used is CSP4J [16]. This solver, which competed in the International CSP Solver Competition in 2008 [17], uses XCSP, an XML format used to express constraint networks, as its input format. In comparison to OPL, XCSP is a low level CSP specification language that requires each constraint be defined individually either in extension or in intension. As a result, its encoding length can be very large.

Table 6.1 shows the hardware specification of our test system. More importantly, all experiments are run with data files located on a ramdisk (in memory file system) helping to reduce the cost of I/O. This is relevant because, unlike the other solvers, SCDE requires substantial of interaction with a SqlLite, a lightweight file based database. SCDE also creates pseudo boolean files for the MiniSat+ solver. All of this I/O is expensive even on a ramdisk, but is not done by traditional solvers for which the data extraction and encoding have been done ahead of time.

CPU Name	Intel(R) Xeon(R) CPU X5680
CPU Speed	3.33GHz
Number of Cores	24
Memory	64Gb

Table 6.1: Solve And Translation Times in milliseconds

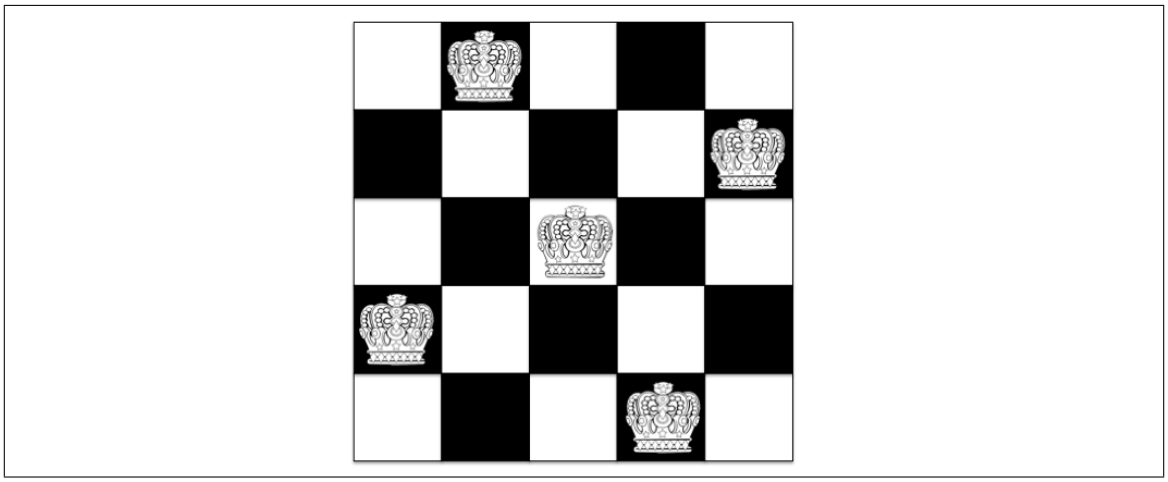

Figure 6.1: The 5-Queens Problem

6.1 NQueens

In this section we analyze the classic n-Queens problem. The problem requires the placement n queens on a chess board according to the following 4 constraints.

1. All queens must be placed on the board.

2. No two queens can share a column.

3. No two queens can share a row.

4. No two queens can share a diagonal.

Figure 6.1 shows example solution to the $n = 5$ problem.

In a traditional CSP solver, the first constraint can be captured automatically, assuming each variable represents the location of a queen, by the notion that all variables must be assigned a value. SQL, however, has no notion of CSP variables and thus this constraint needs to be expressed explicitly. Constraints 2 and 3 can be rewritten as each row and column must have 1 queen due to the fact that there are n queens on an $n \times n$ board. This can allow for several optimizations both in how the solution table is designed and what constraints must be expressed. The last constraint can be enforced by checking that the difference between two queens' row indexes is not equal to the difference between their column indexes.

6.1.1 SCL Specification

In this specification we take advantage of the fact that fixing each queen to a column satisfies constraint 2 automatically and simplifies our solution table. With this in mind we create the two base tables specified in Figure 6.2. The queens.id attribute represents both the identity of a queen and the column it is bound to. The b_rows.id attribute represents the row index.

The constraint schema for this problem is specified in figure 6.3. It creates the relation board, and maps the queens:id attribute to the b_rows:id attribute. It also specifies constraint 1 as $C1$ and constraints 3,4 as $C2$. We note that $C2$'s query is

```
CREATE TABLE queens(id);
    INSERT INTO queens VALUES(1);
    INSERT INTO queens VALUES(2);
    . . .
    INSERT INTO queens VALUES(8);

CREATE TABLE b_rows(id);
    INSERT INTO b_rows VALUES(1);
    INSERT INTO b_rows VALUES(2);
    . . .
    INSERT INTO b_rows VALUES(8);
```

Figure 6.2: Encoding 1: SQL for N-Queens base tables.

```
CREATE SOLUTION TABLE board(
        qid INTEGER FOREIGN KEY REFERENCES queens(id),
        rid INTEGER FOREIGN KEY REFERENCES b_rows(id),

        --Each queen placed once.
        CONSTRAINT C1 CHECK (NOT EXISTS
                (SELECT q.id FROM queens q WHERE
                1 <> (SELECT COUNT(*) FROM board b WHERE b.qid=q.id))),

        --No two queens can attack each other
        CONSTRAINT C2 CHECK (NOT EXISTS
                (SELECT * FROM board b1, board b2 WHERE
                        --distinct pair of queens.
                        b1.qid > b2.qid AND
                        --No two queens share a row.
                        (b1.rid = b2.rid OR
                        --No two queens share a diagonal.
                        (b1.qid - b2.qid) = ABS(b1.rid - b2.rid))))
);
```

Figure 6.3: Encoding 1: SCL for N-Queens problem

expensive, with respect to translation, as it requires a join of 2 copies of the constraint schema. As constraint schemas are replaced by variable maps at execution time, and variable maps are typically large, this can result in a large SQL query run time. On the other hand, the encoding for the problem is relatively succinct, and thus leads credence to SCL as a viable CSP specification language.

Constraint $C1$ is an efficient encoding of the constraint "All queens must be placed

on the board." For each queen i an eCNF expression is generated which has the form $\{v_{i,1}, \ldots, v_{i,j}, \ldots, v_{i,n}\} = 1$ where j is the row index of the queen. If we target a SAT solver that requires CNF form, then this eCNF expression must be expanded to a set of clauses of the form $(\neg v_a \vee \neg v_b)$ where a, b represent all unique pairs of variables from the original eCNF clause, and 1 clause of the form $(v_{i,1} \vee \ldots \vee v_{i,n})$. Thus, for all n queens, we end up with $n \times (\binom{n}{2} + 1)$ CNF clauses. As an alternative, constraint $C1$ could be encoded by the SQL below.

```
--n queens on the board.
CONSTRAINT C1 CHECK
        (8 = (SELECT COUNT(*) FROM board b)),
```

This encoding may be appealing to an RDB practitioner as it directly expresses the goal of putting n queens on the board. Furthermore, with a slight modification to constraint $C2$ which forbids the same queen from being placed twice, this encoding can work. An analysis of this encoding, however, reveals an important negative impact on the resulting CNF encoding and overall solve time. The constraint generates only 1 eCNF expression which has the form $\{v_{1,1}, \ldots, v_{i,j}, \ldots, v_{n,n} = n\}$ where v_{ij} represents queen i in row j. Thus this clause has n^2 variables. In CNF form the constraint expands to $\binom{n^2}{(n+1)}$ clauses of the form $(\neg v_1 \vee \ldots \vee \neg v_{n+1})$ and $\binom{n^2}{(n-1)}$ clauses of the form $(v_1 \vee \ldots \vee v_{n-1})$. Thus the total amount of clauses needed is $\binom{n^2}{(n+1)} + \binom{n^2}{(n-1)}$, which, as an example, when $n = 80$ is over 2×10^{187} clauses. Thus this constraint encoding is unscalable for CNF based SAT solvers. For example, at

```
using CP ;
int n = 8;
range Queen_Ids = 1..n ;
range Row_Ids = 1..n ;

// Decision Variable
dvar int queens[Queen_Ids] in Row_Ids;

// Solving
subject to {
        forall(i,j in Queen_Ids: j > i) { // distinct pair of queens.
                 // No two queens share a row.
                queens[i] != queens[j] ;
                // No two queens share a diagonal.
                abs(queens[j] - queens[i]) != (j - i) ;
        };
};
```

Figure 6.4: OPL specification for N-Queens problem.

$n = 80$, MiniSat+'s solve time for the first encoding is about 1 second, compared to 4 seconds for the second encoding.

6.1.2 OPL and XCSP Specification

In Figure 6.4 we give an OPL specification for the n-Queens problem. Note that there may be more effective specifications in OPL, but we strive to mimic the SCL specification's constraints as closely as possible.

In Figure 6.5 we give the same specification in XCSP. Note that a separate constraint must be specified for each pair of queens, but all of these constraints leverage the same predicate, $P0$, enforcing the diagonal condition.

```
<instance>
        <presentation name="NQueens" maxConstraintArity="2" format="XCSP_2.0"/>
        <domains nbDomains="1">
                <domain name="D0" nbValues="8">1..8</domain>
        </domains>
        <variables nbVariables="8">
                <variable name="V0" domain="D0"/>
                . . .
                <variable name="V7" domain="D0"/>
        </variables>
        <predicates nbPredicates="1">
                <predicate name="P0">
                        <parameters>int X0 int X1 int X2</parameters>
                        <expression>
                                <functional>
                                        and(ne(X0,X1),ne(abs(sub(X0,X1)),X2))
                                </functional>
                        </expression>
                </predicate>
        </predicates>
        <constraints nbConstraints="28">
                <constraint name="C0" arity="2" scope="V0 V1" reference="P0">
                        <parameters>V0 V1 1</parameters>
                </constraint>
                        . . .
                <constraint name="C27" arity="2" scope="V6 V7" reference="P0">
                        <parameters>V6 V7 1</parameters>
                </constraint>
        </constraints>
</instance>
```

Figure 6.5: XCSP specification for N-Queens problem.

n	SCDE: Translation	SCDE:MiniSat+ Total	CPLEX Total	CSP4J Total
8	152	153	14	262
40	1864	1965	81	1149
80	18379	19122	351	3224
120	86716	89233	2441	9862

Table 6.2: Translation and Total Solve Times in milliseconds

6.1.3 Results

Table 6.2 gives the translation and total solve times in milliseconds for N-Queens problem for SCDE with the default MiniSat+ engine, as well as the solve times for CPLEX and CSP4J. The bulk of SCDE's work is in translation. The large translation time results from I/O based database operations for data access that other solvers are absent in. Furthermore, the need for expensive operations like joins does not arise in OLP and XCSP due to their explicit definitions of variables, domains, and predicates over them. These advantages typically come at the cost of having to do manual data extraction and preprocessing to generate the mathematical encoding.

The solve times for SCDE's three supported cardinality SAT solvers is given in table 6.3. This table reveals that the boolean encoding generated by SCDE is very effective for a variety of SAT solving approaches. We note that KWSAT shows erratic solve times and assume it is related to it's random variable selection strategy.

In conclusion we note that SCDE shows usable total response times (a few seconds) for $n < 80$, but suffers a large translation penalty for higher n. We also note

n	SCDE: MiniSat+	SCDE: KWSAT	SCDE: KDPLL
8	1	6	3
40	101	67	47
80	743	716	239
120	2517	294	323

Table 6.3: SCDE's SAT Solver Times in milliseconds

that the boolean encoding generated by SCDE can be solved very effectively even for large n, with the $n = 120$ solve time ≈ 0.3 seconds, compared to CPLEX's 2.4 seconds and CSP4J's 9.9 seconds. We believe that some database optimization in the form of indexes could significantly improve our translation time, but have not verified this postulate.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7
period 1	1 v 2	1 v 3	5 v 8	4 v 7	4 v 8	2 v 6	3 v 5
period 2	3 v 4	2 v 8	1 v 4	6 v 8	2 v 5	1 v 7	6 v 7
period 3	5 v 6	4 v 6	2 v 7	1 v 5	3 v 7	3 v 8	1 v 8
period 4	7 v 8	5 v 7	3 v 6	2 v 3	1 v 6	4 v 5	2 v 4

Table 6.4: RoundRobin solution for $n = 8$

6.2 Round Robin Tournament

In this section we analyze the RoundRobin problem. The problem is to schedule a tournament of n teams over $n - 1$ weeks, with each week divided into $n/2$ periods, and each period has one game divided into two slots. The first slot is for the team playing at home, while the second is for the team playing away. A tournament must satisfy the constraints listed below.

1. Every team plays once a week.

2. Every team plays every other team once.

3. Every team plays at most twice in the same period over the tournament.

A solution to the $n = 8$ problem is given in Table 6.4.

Note that swapping the home and away teams is harmless because no constraint directly references home or away. This allows us to add the extra constraint that the home team id (hid) must always be less (or greater) than the away team id (aid)

for any game without altering the problem. This extra constraint drastically reduces the search space. We note that the total number of games required such that each team plays every other team will be:

$$\binom{n}{2} = \frac{n * (n-1)}{2}$$

This is exactly the total number of periods in our schedule, so a solution to the problem will require all periods be filled with a unique combination of teams, and all unique combinations (with $hid < aid$) be used. Even though there is no theoretical difference between using the constraint $hid < aid$ or $hid > aid$, both solvers tested (SCDE and CPLEX OPL) showed significantly different runtimes. This is most likely due to internal variable ordering. As a result, the best performing option was taken for each solver.

6.2.1 SCL Encoding

The encoding for this problem begins with a set of base tables, given in Figure 6.6, that provide the domains for the attributes in the constraint schema. The constraint schema combines the weeks.id, periods.id, teams.id, and teams.id attributes as foreign keys mapped to week, period, home, and away respectively. For decomposition purposes, we designate the key {home, away}, which is valid due to constraint 2. The SCL specification is given in Figure 6.7.

```
CREATE TABLE teams(id);
        INSERT INTO "teams" VALUES(1);
        INSERT INTO "teams" VALUES(2);
        . . .
        INSERT INTO "teams" VALUES(8);

CREATE TABLE weeks(id);
        INSERT INTO "weeks" VALUES(1);
        INSERT INTO "weeks" VALUES(2);
        . . .
        INSERT INTO "weeks" VALUES(7);

CREATE TABLE periods(id);
        INSERT INTO "periods" VALUES(1);
        INSERT INTO "periods" VALUES(2);
        INSERT INTO "periods" VALUES(3);
        INSERT INTO "periods" VALUES(4);
```

Figure 6.6: SQL RoundRobin base tables.

6.2.2 CB-Decomposition

Our automated decomposition searches for constraints that reference a subset of the constraint schema attribute set and creates new constraint schemas for those subsets. Each new schema must contain the designated key {home, away}. The algorithm creates the following three constraint schemas.

- Sub1 containing attributes home, away.
- Sub2 containing attributes week, home, away.
- Sub3 containing attributes period, home, away.

```
CREATE CONSTRAINT TABLE schedule (
        week INTEGER FOREIGN KEY REFERENCES weeks(id),
        period INTEGER FOREIGN KEY REFERENCES periods(id),
        home INTEGER FOREIGN KEY REFERENCES teams(id),
        away INTEGER FOREIGN KEY REFERENCES teams(id),

        KEY (home, away),

        --no team can play itself.
        CONSTRAINT C0 CHECK ( NOT EXISTS
                (SELECT * FROM schedule s WHERE s.home <= s.away)),

        --Each period must happen once a week.
        CONSTRAINT C1 CHECK ( NOT EXISTS
                (SELECT w.id, p.id FROM weeks w, periods p WHERE
                1 <> (SELECT COUNT(*) FROM schedule s
                        WHERE s.week = w.id AND s.period=p.id))),

        --every team plays once a week.
        CONSTRAINT C2 CHECK ( NOT EXISTS
                (SELECT w.id, t.id FROM weeks w, teams t WHERE
                1 <> (SELECT COUNT(*) FROM schedule s
                        WHERE s.week = w.id AND
                        (s.home = t.id OR s.away = t.id)))),

        --every team plays at most twice in the same period.
        CONSTRAINT C3 CHECK ( NOT EXISTS
                (SELECT p.id, t.id FROM teams t, periods p WHERE
                2 < (SELECT COUNT(*) FROM schedule s
                        WHERE s.period = p.id AND
                        (s.home = t.id OR s.away = t.id)))),

        --every team plays every other team once.
        CONSTRAINT C4 CHECK ( NOT EXISTS
                (SELECT t1.id, t2.id FROM teams t1, teams t2
                WHERE t2.id < t1.id AND
                1 <> (SELECT COUNT(*) FROM schedule s
                        WHERE s.home = t1.id AND s.away = t2.id)))
);
```

Figure 6.7: SCL for RoundRobin problem

The constraints are then remapped to new schemas as follows.

- $C0, C4$ mapped to Sub1.
- $C2$ mapped to Sub2.
- $C3$ mapped to Sub3.
- $C1$ mapped to join of Sub2 and Sub3.

Each of the new constraint schemas have significantly smaller VBMaps than the original schema. This decreases the search space for satisfying each constraint, which in turn saves on solve time.

Note the choice of key is important. For example, {week,period} is also a valid key. Unfortunately it does not work for our implementation because constraints $C0, C2, C3, C4$ would generate new constraint schemas based on $\mu(C|\text{schedule}) \cup$ {week,period} which contains all the attributes of the original schema schedule. Our implementation requires any new constraint schemas to be strict subsets of the original constraint schema in order to force a reduction in VBMap sizes.

6.2.3 OPL and XCSP Encodings

In this section we begin by exploring an OPL specification for the RoundRobin problem similar to the SCL specification given in Figure 6.7. The OPL encoding, given in

Figure 6.8, creates one decision variable for every combination of $\{week, home, away\}$ with corresponding domain $\{0..nbPeriods\}$ where 0 specifies that the given match-up is not played for the given week. Given this, some constraints are more easily expressed as duals. For example the constraint "No team can play twice in the same week" forces all but one of the variables for a given team on a given week to the value of 0.

The XCSP encoding for the RoundRobin problem is much more challenging due to the fact that CSP4J does not support a "count" type constraint. Below we discuss two different XCSP specifications for the $n = 6$ problem for which CSP4J fails to find a solution in a reasonable amount of time.

XCSP Specification 1

In our first approach, we create one variable G_{wp} for each combination of week, period. There are $\frac{n*(n-1)}{2} = 15$ G variables for $n = 6$, and each has a domain $\Delta(G)$ with size $\binom{n}{2} = 15$. Each value in $\Delta(G)$ corresponds to a unique pair of teams {hid,aid} with hid $<$ aid as seen in table 6.5. Given this, the constraint "Every team plays every other team" simply requires all variables to be assigned to unique values. This constraint is commonly referred to as the "AllDifferent" constraint.

The constraint "Every team plays once a week" can be modeled as "No team can play twice in one week" because each team must play $n - 1$ games and there are

```
using CP ;

int nbTeams = 8;
int nbWeeks = nbTeams-1;
int nbPeriods = nbTeams div 2;
int nbGames = nbWeeks*nbPeriods;

range Teams = 1..nbTeams;
range Weeks = 1..nbWeeks;
range Periods = 0..nbPeriods; //where 0 is no game played.

dvar int schedule[Weeks, Teams, Teams] in Periods;

subject to {
        //home team always smallest team / no team can play it self
        forall(h,a in Teams: h>=a) {
                forall(w in Weeks) schedule[w,h,a]==0;
        };

        //Each period happens once a week.
        forall(p in Periods: p>0, w in Weeks){
                count( all(h,a in Teams: h<a)  schedule[w,h,a], p)==1;
        };

        //No team can play twice in the same week.
        forall(t in Teams, w in Weeks) {
                //(period must be 0 in all others).
                count( all(h,a in Teams: h<a && (h==t || a==t))
                        schedule[w,h,a], 0)==nbWeeks-1;
        };

        //No team can play more than 2 times in the same period.
        forall(t in Teams, p in Periods: p>0) {
                count( all(h,a in Teams: h<a && (h==t || a==t), w in Weeks)
                        schedule[w,h,a], p)<=2;
        };

        //Each team must play every other team once.
        forall(h,a in Teams: h<a) {
                //(period must be 0 in all other cases).
                count( all(w in Weeks)  schedule[w,h,a], 0)==nbWeeks-1;
        };
};
```

Figure 6.8: OPL for RoundRobin problem

$n-1$ weeks. One way to express this constraint is via "extension" by listing all pairs of values in $\Delta(G)$ representing two matches with the same team, and then requiring that no two variables G_{wa}, G_{wb} for the week w may be simultaneously assigned to these pairs of values. Table 6.6 gives an example relation for team $id = 2$ in the $n = 6$ problem. Since each team has $n-1$ possible matches, there are $n \times \binom{(n-1)}{2} = 60$ tuples in the complete version of the relation. As there are $n-1$ weeks, $n/2$ variables per week, and all pairs (regardless of order) must be considered, the total number of clauses of the form $G_{wa} \neq X \vee G_{wb} \neq Y$ is $(n-1) \times ({}^{(n/2)}\mathrm{P}_2) = 30$ (where P is the permutation operator) for each tuple in the relation. The final amount of clauses for this constraint is given below.

$$n \times \binom{(n-1)}{2} \times (n-1) \times {}^{(n/2)}\mathrm{P}_2 = 1800$$

The constraint "Every team plays at most twice in the same period over the tournament" can be modeled similarly to the second constraint. A relation is formed listing all triples of $G-values$ representing 3 matches with the same team, and then requiring that no three variables for the same period may simultaneously be assigned to these triplets of values. Given that there are $n/2$ periods and $n-1$ variables for each period, the total number of clauses of the form $G_i \neq v_x \vee G_j \neq v_y \vee G_k \neq v_k$ is given by the formula below.

$$n \times \binom{(n-1)}{3} \times (n/2) \times {}^{(n-1)}\mathrm{P}_2 = 10800$$

$\Delta(G)$	home	away
1	1	2
2	1	3
3	1	4
4	1	5
5	1	6
6	2	3
7	2	4
8	2	5
9	2	6
10	3	4
11	3	5
12	3	6
13	4	5
14	4	6
15	5	6

Table 6.5: Domain of G variables

G_{wa}	G_{wb}
1	6
1	7
1	8
1	9
6	7
6	8
6	9
7	8
7	9
8	9

Table 6.6: Illegal combinations for team 2

XCSP Specification 2

In our second approach, we create one variable for each combination of $\{\mathsf{week},\mathsf{home},\mathsf{away}\}$ called V_{wha}. There are $(n-1) \times \binom{n}{2} = 75$ variables for $n = 6$, and the domain of these variables is the periods $\{0..n/2\}$ where 0 specifies that the given match up is not played for the given week.

This model requires a few extra constraints imposing the correct periods per week. The first of these constraints requires that the $\sum_{h=1}^{n} \sum_{a=h+1}^{n} V_{wha} = \sum_{p=1}^{n/2} p$

for any week w, where p represents the periods. The second constraint forbids any period (besides 0) from occurring twice in the same week and is expressed by the clause $(V_1 == 0 \vee V_2 == 0 \vee V_1! = V_2)$ where V_1, V_2 represent all pairs of variables V_{wha} for week w. Given that there are $\binom{n}{2}$ variables for any week w, the formula below gives the total number of clauses of this form.

$$(n-1) \times \binom{(\binom{n}{2})}{2} = 5 * \binom{15}{2} = 525$$

The constraint "Every team plays once a week" can be modeled as "No team can play twice in one week" because each team must play $n-1$ games and there are $n-1$ weeks. We model this constraint with the clause $(V_1 == 0 \vee V_2 == 0)$ for all pairs of variables V_{wha} where $h == t \vee a == t$ and t, w represent a specific team and week. For a specific week w and team t there are $n-1$ variables V_{wha}. The formula below gives the total number of clauses of this form.

$$(n-1) \times n \times \binom{(n-1)}{2} = 300$$

The constraint "Every team plays every other team" is expressed with a clause of the form $(V_1 \vee V_2 \vee ...V_i...V_{(n-1)\times(n/2)})$ where V_i are variables representing every week and period a specific combination of $home, away$ teams can play in. As there are $\binom{n}{2}$ team combinations, the total number of clauses of this form is given by the

formula below.

$$\binom{n}{2} = 15$$

The constraint "Every team plays at most twice in the same period over the tournament" is expressed with a clause of the form $(V_1 == 0 \vee V_2 == 0 \vee V_3 == 0 \vee V_1! = V_2 \vee V_1! = V_3 \vee V_2! = V_3)$ for all triplets of variables $V_w ha$ where $h == t \vee a == t$ and t represents a specific team. As there are $n-1$ opponents and $n-1$ weeks, there are $(n-1) \times (n-1)$ variables V_{wha} for each team. Given this, the total number of clauses of this form is calculated by the formula below. Luckily we can safely remove any combinations that refers to games in the same week, reducing this number to 7500 for $n = 6$.

$$n \times \binom{(n-1)^2}{3} = 13800$$

Neither of the two encodings presented here are efficient. They highlight some of the complexities in modeling such a problem manually, which contrasts with the SCDE's approach of not using explicit variables or domains. Furthermore, with our decomposition approach, SCDE's underlying variable selection can be difficult to manually duplicate due to the designated key joining mechanism described in Section 4.2.4. In summary, Table 6.7 shows the resulting run times of these specifications for $n = 6$ problem with the CSP4J engine. As these times are much larger than the run times of OPL and SCDE, the XCSP specification is not used in the final results

XCSP Spec 1	XCSP Spec 2	SCDE:Regular
>2400 sec	488.47 sec	0.45 sec

Table 6.7: Total Solve Times for CSP4J at $n = 6$

	Translation Time		Solve Time	
n	Regular	Decompose	Regular	Decompose
10	0.78 sec	1.71 sec	0.21 sec	0.97 sec
12	1.01 sec	3.69 sec	57.24 sec	7.62 sec
14	1.72 sec	8.89 sec	588.76 sec	10.99 sec
16	2.2 sec	20.85 sec	>144000 sec	140.21 sec

Figure 6.9: Experiment 2 Translation/Solve Times

section.

6.2.4 Results

Figure 6.9 shows the SCDE translation and solve times for the problem. As the run time is dominated by the exponentially growing solve time for $n > 10$, the translation time is usually insignificant. As many problems have this property, we expect SCDE's translation time to be a non-issue in the general case.

Figure 6.10 compares the SCDE run times to the CPLEX run times based on the OPL specification of Figure 6.8. As the OPL specification does not decompose the problem, it provides a basis for comparison to our automated decomposition approach. The results highlight the value of the CB-Decomposition algorithm in improving simple specifications. We note that the manual decomposition used in

n	SCDE:Regular	SCDE:Decomposed	CPLEX
10	0.99 sec	2.68 sec	0.107 sec
12	58.25 sec	11.31 sec	3.18 sec
14	590.48 sec	19.88 sec	51.26 sec
16	>144000 sec	161 sec	130392 sec

Figure 6.10: Experiment 2 Run Times

the OPL specification in [15] achieves sub minute runtimes for the RoundRobin problem through $n = 30$. This specification leverages arc-consistency and other sophisticated techniques which are, in our opinion, outside the expertise of the typical RDB practitioner.

6.3 Course Scheduling

In this section we analyze a more complex version of the university course scheduling problem described in Figure 3.1. The constraint schema creates a schedule that relates courses, professors, teaching times, and rooms/facilities. Experimental results show averages from 24 separate data instances with some randomized elements. Each instance has 7 time slots, 4 rooms, 28 sections of 10 different courses, and 15 professors. The number of sections a professor is contracted to teach is randomly assigned and must be met in the solution, balanced by the requirement that all sections must be taught. Other random elements include matching qualifications of professors to courses, what rooms/facilities meet the needs of each course, preferences for time slots, and whether a professor will teach in adjacent time slots (if more than one section is contracted).

6.3.1 SCL Specification

Two variations of the problem are used. Experiment **A** has 10 constraints, each referencing 3 or less of the 4 attributes in the constraint schema schedule. The SCL for experiment **A** is given in Figure 6.11. Experiment **B** adds an 11th constraint which requires professors to remain in the same room if they teach in adjacent time slots. This constraint, given in Figure 6.12, references both attributes outside of

```
CREATE CONSTRAINT TABLE schedule(
   cid INTEGER FOREIGN KEY REFERENCES courses(id),
   pid INTEGER FOREIGN KEY REFERENCES professors(id),
   rid INTEGER FOREIGN KEY REFERENCES rooms(id),
   tid INTEGER FOREIGN KEY REFERENCES time_slots(id),
   KEY(cid, tid),

   --No professor can teach outside of his course list.
   CONSTRAINT F1 CHECK (NOT EXISTS
     (SELECT * FROM schedule s, courses c WHERE s.cid=c.id AND c.name NOT IN (
       SELECT pc.cname FROM prof_courses pc WHERE pc.pid = s.pid))),

   --No professor can teach before or after his early/late times.
   CONSTRAINT F2 CHECK (NOT EXISTS
     (SELECT * FROM schedule s, professors p WHERE
     s.pid = p.id AND (s.tid<p.early_slot OR s.tid>p.late_slot))),

   --Each prof teaches num times.(p)
   CONSTRAINT C3 CHECK (NOT EXISTS
     (SELECT p.id, p.num FROM professors p WHERE
     p.num <> (SELECT COUNT(*) FROM schedule s WHERE s.pid=p.id))),

   --No professor can teach multiple times during the same time slot.
   CONSTRAINT C4 CHECK (NOT EXISTS
     (SELECT p.id, t.id FROM professors p, time_slots t WHERE
     1 < (SELECT COUNT(*) FROM schedule s WHERE s.pid = p.id AND s.tid=t.id))),

   --Some professors can NOT teach Back to Back (BB=no).
   CONSTRAINT C5 CHECK (NOT EXISTS
     (SELECT * FROM schedule s1, schedule s2, professors p WHERE
     s1.pid=s2.pid AND p.id=s1.pid AND s1.tid>s2.tid AND
     p.back_to_back='No' AND (s1.tid - s2.tid)=1)),

   --Each course must be taught exactly once.
   CONSTRAINT C6 CHECK (NOT EXISTS
     (SELECT c.id FROM courses c WHERE
     1 <> (SELECT COUNT(*) FROM schedule s WHERE s.cid = c.id))),

   --No room can be booked more than once per time slot.
   CONSTRAINT C7 CHECK (NOT EXISTS
     (SELECT r.id, t.id FROM rooms r, time_slots t WHERE
     1 < (SELECT COUNT(*) FROM schedule s WHERE s.rid=r.id AND s.tid=t.id))),

   --Each course must be in room big enough
   CONSTRAINT F8 CHECK (NOT EXISTS
     (SELECT * FROM schedule s, rooms r, courses c WHERE
     s.rid = r.id AND s.cid = c.id AND c.size>r.size)),

   --Course with projector requirements must be in appropriate rooms.
   CONSTRAINT F9 CHECK (NOT EXISTS
     (SELECT * FROM schedule s, rooms r, courses c WHERE
     s.rid = r.id AND s.cid = c.id AND c.projector='Yes' AND r.projector='No')),

   --Course Pairs may not be taught at the same time.
   CONSTRAINT C10 CHECK (NOT EXISTS
     (SELECT * FROM schedule s1, schedule s2, course_pairs cp WHERE
     s1.cid=cp.cid1 AND s2.cid=cp.cid2 AND s1.tid=s2.tid))
);
```

Figure 6.11: SCL for Course Scheduling Problem

the designated key. As a result, the automated decomposition must join several constraint schemas to process the constraint (see Section 4.2.4).

```
CONSTRAINT C11 CHECK (NOT EXISTS
  (SELECT * FROM schedule s1, schedule s2 WHERE
  s1.pid=s2.pid AND s1.tid>s2.tid AND (s1.tid - s2.tid)=1 AND s1.rid<>s2.rid))
```

Figure 6.12: SCL for Constraint 11 - Experiment **B**

6.3.2 Results

We begin by examining the impact of CB-decomposition on the final boolean encoding metrics. Figure 6.13 compares the number of variables, the number of clauses, and the total size of the encoding (reflecting more globally the size of the clauses). These properties give useful insights into the potential speed up for both the translation time from SQL to SAT, and the solve time for the boolean solver.

We note that the number of boolean variables for both experiments is identical, as the tuples associated with the relations courses, professors, times and rooms are identical. The decomposed version of the problem has fewer variables as not all

Metric	Exp	Original	Decomposition
Num of Variables	A	2786	1710
	B	2786	1710
Num of Clauses	A	108223	7728
	B	193226	92731
Encoding Size	A	2693 KB	216 KB
	B	4746 KB	3633 KB

Figure 6.13: Encoding Metrics for Course Scheduling

combinations of the four attributes are associated with a boolean variable. Only certain subsets, for instance professors.id and courses.id have unique boolean variables.

Experiment **A** shows significantly fewer boolean clauses generated for the decomposed problem than the original. This is expected as constraints reference smaller variable maps. In experiment **B**, the 11th constraint alone, which must reference the entire schedule schema, accounts for 85,003 of the 92,731 clauses.

Figures 6.14 and 6.15 show the run times in seconds for the translation and solving steps for the two experiments. As experiment **A** has no constraints across all attributes of the solution table, it's decomposition produces more independent sub-problems which are easily translated and solved. The result is a dramatic ($\approx 4.5x$) improvement in total run time. Experiment **B**, on the other hand, contains a constraint that requires a join of sub-solution tables when decomposed. This constraint has a much larger translation time than the time it takes to translate the original problem. Furthermore, the sub-solutions are not independent. This results in a weaker decomposition with a modest performance improvement. We anticipate that, for problems of this type, CB-decomposition will reduce the solve time but will incur a higher translation time. Hence the overall value of the decomposition will increase as the average solve time becomes the dominant factor in the total time to solution.

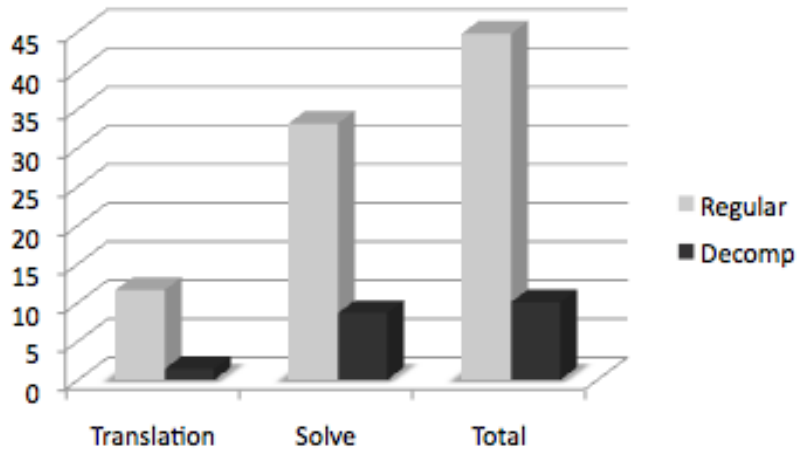

Figure 6.14: Experiment **A** Run Times (sec)

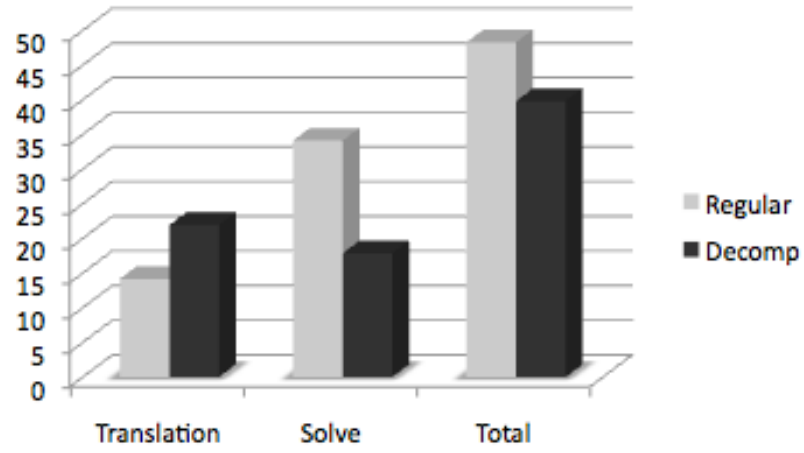

Figure 6.15: Experiment **B** Run Times (sec)

6.4 Oxford College Freshman Partitioning

In this section we analyze a partition style problem posed to us by Oxford College for their incoming freshman class in 2009. The problem requires 400 students to be split into 25 groups, of size 16 each, while maximizing the even distribution of 3 dimensions; gender, geographic origin, and ethnic diversity. More precisely the minimum and maximum number of students in any one group belonging to a particular category within a dimension should be maximized and minimized respectively. For example, if there are 26 students with a geographic origin of Latin America, then the ideal solution would have 2 of these students in 1 group, and 1 student in each of the remaining 24 groups.

Mathematically, let set $\mathcal{G} = \{H_1, \ldots, H_n\}$ be the set of all students where each student entry H consists of a vector of dimension 3, $H = \{H[d_1], H[d_2], H[d_3]\}$, representing a value in each of the three dimensions $\{d_1, d_2, d_3\}$ with corresponding domains $\{\Delta(d_1), \Delta(d_2), \Delta(d_3)\}$. Let $|Y[d_i] = x|$ be the number of students in set Y with $H[d_i] = x$. The problem is to partitioned $\mathcal{G}$ into sets $G_1, \ldots, G_m$ of equal size such that the $\min_{1 \leq k \leq m} |G_k[d_i] = x|$ is maximized and $\max_{1 \leq k \leq m} |G_k[d_i] = x|$ is minimized for all $d_i \in \{d_1, d_2, d_3\}$, $x \in \Delta(d_i)$. Note in our problem $n = 400, m = 25, |\Delta(d1)| = 2, |\Delta(d2)| = 8, |\Delta(d3)| = 7$ where d_1 is gender, d_2 is geographic location, and d_3 is ethnic diversity.

6.4.1 SCL Specification

As SCDE does not directly support optimization, we replace the optimization constraints with satisfaction constraints:

$$|G_k[d_i] = x| \leq \lceil |\mathcal{G}[d_i] = x|/m \rceil$$

$$|G_k[d_i] = x| \geq \lfloor |\mathcal{G}[d_i] = x|/m \rfloor$$

where $|\mathcal{G}[d_i] = x|$ is the total number of students with $H[d_i] = x$ and $\lceil z \rceil$, $\lfloor z \rfloor$ are the ceiling and floor functions respectively. We note that for our particular data set there exists an optimal solution where every constraint can be satisfied.

The SCL specification references a database comprised of 5 tables including geocode, diversity, and gender, containing the domains $\Delta(d_1), \Delta(d_2)$ and $\Delta(d_3)$ respectively, the group table containing the id of each of the 25 groups $\{G_1, \ldots, G_{25}\}$, and the people(id, geocode, diversity, gender) table containing all 400 students in $\mathcal{G}$. The constraint schema maps the people.id to the group.id attribute. The SCL is given in Figure 6.16. Note that the constraint right-hand-side values are pre-calculated and stored in the geocode.max/min, diversity.max/min, and gender.max/min attributes.

```
CREATE CONSTRAINT TABLE studgroups(
   pid INTEGER FOREIGN KEY REFERENCES PEOPLE(id),
   gid INTEGER FOREIGN KEY REFERENCES GROUPS(id),

   --Each student must be in exactly 1 group.
   CONSTRAINT C0 CHECK (NOT EXISTS
   (SELECT p.id FROM people p WHERE
     1 <> (SELECT COUNT(*) FROM studgroups s WHERE s.pid = p.id))),

   --Each group must have 16 people total
   CONSTRAINT C1 CHECK (NOT EXISTS
   (SELECT g.id FROM groups g WHERE
     16 <> (SELECT COUNT(*) FROM studgroups s WHERE s.gid = g.id))),


   --Each group should have gender balance.
   CONSTRAINT C_GENDER_MAX CHECK ( NOT EXISTS
   (SELECT  grp.id, gen.type, gen.max FROM groups grp, gender gen
     WHERE gen.max < (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.gender = gen.type))),

   CONSTRAINT C_GENDER_MIN CHECK ( NOT EXISTS
   (SELECT  grp.id, gen.type, gen.min FROM groups grp, gender gen
     WHERE gen.min > (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.gender = gen.type))),


   --Each group should have geocode balance.
   CONSTRAINT C_GEOCODE_MAX CHECK ( NOT EXISTS
   (SELECT  grp.id, g.type, g.max FROM groups grp, geocode g
     WHERE g.max < (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.geocode = g.type))),

   CONSTRAINT C_GEOCODE_MIN CHECK ( NOT EXISTS
   (SELECT  grp.id, g.type, g.min FROM groups grp, geocode g
     WHERE g.min > (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.geocode = g.type))),


   --Each group should have diversity balance.
   CONSTRAINT C_DIVERSITY_MAX CHECK ( NOT EXISTS
   (SELECT  grp.id, g.type, g.max FROM groups grp, diversity g
     WHERE g.max < (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.diversity = g.type))),

   CONSTRAINT C_DIVERSITY_MIN CHECK ( NOT EXISTS
   (SELECT  grp.id, g.type, g.min FROM groups grp, diversity g
     WHERE g.min > (SELECT  COUNT(*) FROM studgroups s, people p
       WHERE s.pid = p.id AND s.gid = grp.id AND p.diversity = g.type)))
);
```

Figure 6.16: SCL for Oxford Freshman Partitioning problem

6.4.2 OPL Specification

In this section we give an OPL specification for the Oxford freshman partitioning problem. The specification, given in Figure 6.17, is similar to the SCL specification. The data arrays gender, geocode, and diversity contain a copy (abbreviated in the figure for readability) of the original database. The decision variable array assigns each student to 1 group, implicitly enforcing the first SCL constraint. The remaining constraints are similar to the SCL specification.

6.4.3 Results

Figure 6.18 shows the total run times of the problem for SCDE with all 3 backend solvers, and the CPLEX solver. The results demonstrate the importance of supporting a variety of backend solvers and provides more evidence for the feasibility of our approach.

Solver	Run Time
SCDE with MiniSat+	$>$ 900 sec
SCDE with KDPLL	$>$ 900 sec
SCDE with KWSAT	23.6 sec
CPLEX	293.0 sec

Figure 6.18: Run Times for Oxford Freshman Partitioning problem

```
using CP ;

int nbStudents = 400;
int nbGroups = 25;
int nbGrpSize = nbStudents div nbGroups;
int nbGenders = 2;
int nbGeoCodes = 8;
int nbDiversity = 7;

range rStud = 1..nbStudents;
range rGrp = 1..nbGroups;
range rGen = 1..nbGenders;
range rGeo = 1..nbGeoCodes;
range rDiv = 1..nbDiversity;

int gender[rStud] = [2,2,2,1,2,2,1,1,2,1,1,2,1,2,2,1,2,1,1,1...
int genderMax[rGen] = [7, 10];
int genderMin[rGen] = [6, 9];

int geocode[rStud] = [1,4,2,2,1,8,2,1,7,1,1,1,2,4,1,2,2,1,1,1,...
int geocodeMax[rGeo] = [8,4,2,1,1,1,1,1];
int geocodeMin[rGeo] = [7,3,1,0,0,0,0,1];

int diversity[rStud] = [6,6,6,2,3,3,5,6,6,6,3,6,2,4,6,6,2,5,5,3,...
int diversityMax[rDiv] = [1,5,3,1,2,7,1];
int diversityMin[rDiv] = [0,4,2,0,1,6,0];

// Decision Variable
dvar int studgroups[rStud] in rGrp;

// Solving
subject to {
  forall(g in rGrp){
    count( all(s in rStud) studgroups[s], g) == nbGrpSize;
    forall(x in rGen){
      count( all(s in rStud: gender[s]==x) studgroups[s], g) <= genderMax[x];
      count( all(s in rStud: gender[s]==x) studgroups[s], g) >= genderMin[x];
    };
    forall(x in rGeo){
      count( all(s in rStud: geocode[s]==x) studgroups[s], g) <= geocodeMax[x];
      count( all(s in rStud: geocode[s]==x) studgroups[s], g) >= geocodeMin[x];
    };
    forall(x in rDiv){
      count( all(s in rStud: diversity[s]==x) studgroups[s], g) <= diversityMax[x];
      count( all(s in rStud: diversity[s]==x) studgroups[s], g) >= diversityMin[x];
    };
  };
};
```

Figure 6.17: OPL for Oxford Freshman Partitioning problem

Chapter 7

Conclusion

We have described a research effort towards the integration RDB and CSP systems. Our theoretical contributions include a formalization of the RCSP, a scheme for leveraging SQL's DDL for RCSP specification, a set of algorithms for compiling RCSP defined in SQL to boolean satisfiability formulae, and an automated RCSP decomposition technique for reducing solve times. We have created a prototype for testing these techniques and described several implementation-level optimizations which improve overall performance. Finally, we have provided both theoretical and empirical analyses demonstrating the practical value of our efforts.

7.1 Future Work

As this is relatively new research, there remains many unexplored areas worthy of investigation. We list a few key areas that our research has touched on in the past, but never fully addressed.

7.1.1 Optimization Problems

Many practical industry constraint problems require optimization as oppose to satisfaction. In general, optimization problems generalize satisfaction problems by adding cost functions which must be maximized or minimized while maintaining the satisfaction of a set of constraints. A brute force approach, which databases are naturally poised to pursue, is to collect all solutions to the underlying satisfaction problem, and search through them for optimal cost function values. For a majority of problems, this approach is not scalable. Instead, a more complete approach is to develop a flexible technique for cost function specification in relational algebra along with an algorithm for translating these functions into traditional formulae useable by existing optimization systems.

7.1.2 Interactive CSP Solving

Frequent user interaction is a basic premise of dynamic CSPs. Furthermore, there is growing recognition that the ability to incrementally fine-tune constraints is important to fully exploit the power of intelligent tools in solving real-world problems [32]. Although the SCL supports some commands for user interaction, including `TRACE` and `ALTER CONSTRAINT`, these commands are based on static analysis of constraints outside of the solver. An intriguing approach to enabling dynamic CSP interaction is

to more closely integrate the solver and RDB such that changes to relational schema and data dynamically impact active solver work, and feedback from these changes can be viewed in real time.

7.1.3 Other Decomposition Approaches

We have demonstrated a decomposition approach based on attribute partitions determined through constraint analysis. Although this technique has shown promise, many other strategies exist. Preliminary work by [35, 20] shows that domains belonging to certain attributes can be divided into smaller domains creating more easily solved sub-problems. These types of approaches, based on domain decomposition, require further study. In general, automated decomposition is particularly promising for RCSP as rich semantic information is incorporated as part of the problem specification.

7.2 Summary

The benefits of integrating relational databases with constraint satisfaction solvers have always been apparent. A strong integration of these two technologies can be accomplished by specifying CSPs in current RDB languages and tools, and solving these problems via translation to SAT –what Bacchus calls the assembly language of CSPs [38]. Benchmarking reveals this approach is feasible and initial comparison

on a range of test problems shows that even a straightforward encoding, when used in conjunction with an appropriate constraint solver, yields very good performance. Furthermore, performance can be drastically improved for certain problems via automated decomposition. The end result is a viable foundation for a RCSP system that enables typical RDB practitioners to solve CSPs over relational data in an intuitive and efficient manner.

Bibliography

[1] P. Barth. A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003.

[2] F. Bacchus and P. van Beek On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems. 311318, 1998.

[3] J. Biskup *et al.* Solving Equations in the Relational Algebra. *SIAM J. Comput.*, 33(5):1052–1066, 2004.

[4] M. Cadoli and T. Mancini. Combining Relational Algebra, SQL, Constraint Modeling, and Local Search. *TPLP*, 7(1-2):37–65, 2007.

[5] D. Inc. Dwave home page. `http://www.dwavesys.com`.

[6] D. Inc. Dwave sql page. `http://sql.dwavesys.com`.

[7] J. D. Ullman and C. Zaniolo. Deductive databases: achievements and future directions. *SIGMOD Rec.*, 19(4):75–82, 1990.

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress report on the disjunctive deductive database system dlv. In *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 148–163, London, UK, 1998. Springer-Verlag.

[9] M. Cadoli and A. Schaerf. Compiling Problem Specification into SAT. *Artif. Intell.*, 162:89–120, 2005.

[10] D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proc. of DAC*, pp. 830–835, 2003.

[11] H. E. Dixon *et al.* Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *J. Artificial Intelligence Research*, 21:193–243, 2004.

[12] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pp. 502–518, 2003.

[13] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[14] J. P. Walser. Solving linear pseudo-Boolean constraint problems with local search. In *AAAI '97/IAAI '97: Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, pages 269-274.

[15] P. V. Hentenryck, L. Michel, L. Perron and J. C. Rgin Constraint Programming in OPL In *PPDP '99: Proceedings of the International Conference on the Principles and Practice of Declarative Programming*, pages 98-116.

[16] J. Vion CSP4J: a black-box CSP solving API for Java

[17] Third International CSP Solver Competition `http://www.cril.univ-artois.fr/CPAI08/`.

[18] R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In *Complexity of Computation, SIAM-AMS Proc. 7*, pp. 43–73, 1974.

[19] W. Fan *et al.* Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM Trans. Database Syst.*, 33(2):1–48, 2008.

[20] Robin Lohfert and James J. Lu Propositional reasoning by dimensional reduction: a preliminary report. *ACM Southeast Regional Conference*, 525-526, 2007.

[21] A. M. Frisch *et al.* The Rules of Constraint Modeling. *IJCAI*, pp. 109–116, 2005.

[22] M. Gebser *et al.* GrinGo : A New Grounder for Answer Set Programming. In *LPNMR*, 2007.

[23] J. Gil *et al.* Advanced Visual Modeling: Beyond UML. In *ICSE*, pp. 697–698, Orlando 2002.

[24] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer Set Programming Based on Propositional Satisfiability. *J. Autom. Reason.*, 36(4):345–377, 2006.

[25] D. Q. Goldin and P. C. Kanellakis. Constraint Query Algebras. *Constraints*, 1:45-83, 1996.

[26] G. Gottlob, N. Leone and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:2000, 1999.

[27] N. Leone *et al.* The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[28] L. Libkin. *Elements of Finite Model Theory.* Springer, Berlin, 2004.

[29] V. Lifschitz. What Is Answer Set Programming? In *AAAI*, pages 1594–1597, 2008.

[30] G. L. Nemhauser *et al.* Scheduling a Major College Basketball Conference. *Oper. Res.*, 46:1–8, 1998.

[31] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. *CoRR*, cs.AI/0003033, 2000.

[32] B. O'Sullivan. Introduction to the Special Issue on User-Interaction in Constraint Satisfaction. *Constraints*, 9(4):239–240, 2004.

[33] N. Beldiceanu *et al.* Global Constraint Catalogue: Past, Present and Future, 2006

[34] M. Patterson *et al.* Grounding for Model Expansion in k-Guarded Formulas with Inductive Definitions. In *IJCAI*, pp. 161–166, 2007.

[35] Lesi Wang. *An Investigation into Managing SQL-Cardinality Constraints.* Master's thesis, Emory University, 2008.

[36] S. Siva, J. J. Lu, and H. Zhang. A Case Study in Engineering SQL Constraint Database Systems. *ICLP*, pp. 774–778, Udine, Italy, 2008.

[37] H. Zhang. SATO: An Efficient Propositional Prover. *CADE*, pp. 272–275, 1997.

[38] F. Bacchus, CSPs: Adding Structure to SAT. In *SAT*, 10, 2006.