# Extending The Relational Model With Constraint Satisfaction

by

Michael J. Valdron

A thesis submitted to the School of
Graduate and Postdoctoral Studies in
partial fulfillment of the requirements for
the degree of

**Master of Science in Computer Science**

Faculty of Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

January 2021

# Thesis Examination Information

Submitted by: **Michael J. Valdron**

**Master of Science** in **Computer Science**

Thesis title: Extending The Relational Model With Constraint Satisfaction

An oral defense of this thesis took place on January 12, 2021 in front of the following examining committee:

**Examining Committee:**

| | |
|---|---|
| Chair of Examining Committee | Dr. Faisal Qureshi |
| Research Supervisor | Dr. Ken Pu |
| Examining Committee Member | Dr. Akramul Azim |
| Thesis Examiner | Dr. Jeremy Bradbury |

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

# Abstract

We propose a new approach to data driven constraint programming. By extending the relational model to handle constraints and variables as first class citizens, we are able to express first order logic SAT problems using an extended SQL which we refer to as SAT/SQL. With SAT/SQL, one can efficiently solve a wide range of practical constraint and optimization problems. SAT/SQL integrates both SAT solver and relational data processing to enable efficient and large scale data driven constraint programming.

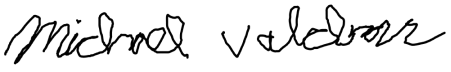
Furthermore, our research presents two novel meta-programming operators: MIN-REPAIR and MIN-CONFLICT which are iterative debugging facilities for constraint programming with SAT/SQL.

**Keywords:** constraints; databases; algebra; optimization; satisfiability

# Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.



Michael J. Valdron

# Statement of Contributions

Part of the work described in Chapter 5 has been published as:

M. Valdron and K. Q. Pu, "Data Driven Relational Constraint Programming," in 2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI), Aug. 2020, pp. 156–163, doi: 10.1109/IRI49571.2020.00030.

I performed the majority of the algorithmic design, experimental evaluation and preparation of the manuscript.

# Acknowledgements

First, I would like to thank my MSc. supervisor Dr. Ken Pu of the Computer Science faculty at Ontario Tech University. Dr. Pu takes great pride and interest in the work we have done together and always strives to ensure I feel the same about what I am working on. Even when his schedule got constricted, he would always make time for meeting with me, whether it was to steer me in the right research direction or mentor me with various challenges I would encounter. Though the unexpected COVID-19 pandemic rendered in-office meeting impossible, he still was able to continue virtually meeting with me consistently.

Second, I would like to thank my committee member, Dr. Akramul Azim, from the Faculty of Engineering at Ontario Tech University and my external examiner, Dr. Jeremy Bradbury, from the Faculty of Science at Ontario Tech University. I am grateful for the valuable feedback on this literature from both Dr. Azim and Dr. Bradbury.

Third and lastly, I would like to express my gratitude to my parents and the rest of my family and friends. All have provided me with unwavering support for the challenges faced with performing research and writing this thesis. This endeavour would not have been possible without their support. Thank you.

Author

Michael J. Valdron

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abbreviations

**1NF** First Normal Form.

**2NF** Second Normal Form.

**3NF** Third Normal Form.

**AI** Artificial Intelligence.

**BCNF** Boyce-Codd Normal Form.

**BCP** Boolean Constraint Propagation.

**CDCL** Conflict Directed Clause Learning.

**CNF** Conjunctive Normal Form.

**CP** Constraint Programming.

**CRM** Constraint Relational Model.

**CRUD** Create Read Update Delete.

**CS** Computer Science.

**CSP** Constraint Satisfaction Problem.

**CSV** Comma Seperated Values.

**DBMS** Database Management System.

**DNF** Disjunctive Normal Form.

**ILP** Integer Linear Programming.

**JSON** JavaScript Object Notation.

**LP** Linear Programming.

**MIP** Mixed Integer Programming.

**NP** Non-deterministic Polynomial.

**OR** Operations Research.

**PB** puseudo-Boolean.

**PL** Propositional Logic.

**REPL** read-evaluate-print-loop.

**RM** Relational Model.

**SAT** Satisfiability.

**SQL** Structured Query Language.

**UNF** Unnormalized Form.

**VSIDS** Variable State Independent Decaying Sum.

# Chapter 1

# Introduction

Constraint Programming (CP) has been a part of the academic research within the area of classical Artificial Intelligence (AI) in Computer Science (CS) since the 1960s [39,47]. Since then there has been a number of research breakthroughs in the area of CP [2,15,36,43]. CP allows one to express complex problems such as those classified in the Non-deterministic Polynomial (NP) complexity space, these are a form of problem known as a Constraint Satisfaction Problem (CSP).

There are a number of CSPs which CP can solve. A well known CSP is the N-Queens problem [45]. Scheduling is another problem that can be modelled as a CSP and can be used in practice. A feasible solution to the scheduling problem would be a complete schedule that contains no conflicts in events and any other constraints that may exist in the particular problem. Another area that uses CSPs to prove or disprove the correctness of algorithms is known as formal verification. There many works which cover CP methods for performing formal verification in detail [10,19,33].

Alongside CP, databases have been a big part of the technology field throughout its evolution. This evolution has also seen the birth of many different kinds

[1]https://www.oracle.com/ca-en/database/

of database systems. The most notable kind of database system is the relational database. Starting with the prototype System R [48] and Oracle DB[1] as the first commercial release in 1979 [29]. With the relational database system came the Structured Query Language (SQL). SQL is the standard language for interfacing directly with relational database systems. Even today, relational database systems are at the forefront of the majority of data-driven systems.

## 1.1 Motivation

The motivation comes in two pillars of problems to solve, structured data management with relational databases and the development in CP for Satisfiability (SAT) solvers.

With the recent rise in big data, it becomes important for database systems to handle complex challenges. One of these challenges is the problem of ensuring the safety and data quality within the contents of database systems. A Database Management System (DBMS) in modern times contains features of limited integrity checking, but it lacks the tasks of optimizing the integrity of the data and ensuring the satisfaction of the constraints within the data. The challenge with incorporating these features by traditional means into a DBMS is due to the computational complexity which is required to perform these tasks, which is NP-complete.

Now we look at what SAT solvers have to play in this. SAT solvers are great solutions to optimization problems which are considered NP-complete. Though there is some work which explores this avenue, such as [34], there has been little done to bring the potential of SAT solvers into the data processing pipeline. In addition, there has been a rise in programming languages using read-evaluate-print-loop (REPL) mechanisms to provide a more iterative and interactive programming experience.

From this, there is motivation to develop a new data processing framework which

merges the technologies of both databases and CP. Due to the popularity of relational database systems, this work focuses on the solution for relational database systems. This framework would present a method of performing SAT solving against database relations. This would involve modification of both how one would interact with the DBMS and the data structure of the relations within the DBMS as well.

To go along with this, we propose improvements to be made to the REPL environment for CP. As we merge SAT solvers and relational DBMSs to show this benefit, so too does it become beneficial to provide an effective REPL experience to the CP aspect of interacting with this theoretical DBMS.

## 1.2 Approach

This section will outline the overall approach taken throughout the thesis research. First, we describe the idea of Data-Driven Constraint Programming. Second, we describe how the idea of extending SQL to include constraint manipulation. Third, we describe the support methods given for iterative and interactive constraint programming.

### 1.2.1 Data Driven Constraint Programming

Data Driven Constraint Programming derives a CP model right from pure data. Lets say we have the following constraints:

$$c_i \leftarrow x > 5 \tag{1.1}$$

$$c_j \leftarrow x \leq y \wedge y \neq 5 \tag{1.2}$$

Where $x$ and $y$ are variables part of CP model. Now lets ask could we express these constraints $c_i$ and $c_j$ as data? Could we create, update, or delete $c_i$ and $c_j$ within a database? The answer is yes to all.

A constraint in itself can be treated as a data structure that defines the type of constraint to enforce and the members to enforce the constraint upon. For instance, $c_i$ is defined as a *greater than* ($>$) constraint with the variable $x$ as the left-hand member and the constant 5 as the right-hand member. To create a constraint is simply to define the components. To update a constraint, we can change the components, such as the case for $c_i$ we could change the constant right-hand member 5 to 3, we could change the constraint type from *greater than* to *less than* ($<$), and so on. To delete a constraint, we can either delete the top-level constraint or a constraint component. For instance, for $c_j$ we could either delete $c_j$ all together or remove the $y \neq 5$ component.

So we can treat constraints as data, how about data as constraints? The answer to this is also yes, however, the solution is more of a complex one. Similar to writing constraints as data, there needs to be able to parse the data into constraints. For instance, if we have a piece of data with *greater than* defined as the constraint type, the variable $x$ as the left-hand component, and 5 as the right-hand component, then this could be parsed into a constraint which enforces $x > 5$. The parser or data reader would need to identify the data structure and which attributes to look for when building the constraint. This would be accompanied by a constraint builder, which would build a constraint from the attributes parsed in whichever CP framework that is used.

In our work, we present a proof of concept framework which meets all the criteria mentioned above. This framework is able to create, update, and delete constraints as data. The framework also can dynamically parse constraints from data with both

predefined and user-defined data structure templates.

### 1.2.2 Constraint Manipulation in SQL

SQL is an efficient, popular, complete, and elegant language for interfacing with relational databases. For these reasons, we would want to reuse what is already a good language. SAT/SQL is what we propose as an extension of the existing SQL with CP added. Additions would include constraint operators and the ability to add CP typed fields.

A concept of this language using relational algebra was proposed and shown in our previous work [51]. In this work, we defined a relational model with constraints. We started by referring to the typical relational model from many relational DBMSs such as PostgreSQL[2]. Then defined the types that would be added to the relational model, these include:

- VAR[$\tau_{\text{data}}$]
- CONSTRAINT

Where VAR[$\tau_{\text{data}}$] is the CP variable type for the relational model and $\tau_{\text{data}}$ is the data type for the CP variable, which would be a type from the traditional relational model. These extended types would be assigned to field within relations and views, which would be containing the CP members. SAT/SQL would allow for querying fields with these extended types familiarly and naturally compared to standard SQL.

Using similar concepts from Section 1.2.1, SAT/SQL can be used to run queries against a capable DBMS to utilize the CP features. For example, one can create a new relation using the `CREATE` statement from SQL with both traditional data types along with the extended type VAR to define CP variable typed attributes, as shown in

[2]https://www.postgresql.org/

```
CREATE TABLE T (
    name TEXT,
    submissions INTEGER,
    groupId? VAR[INTEGER]
);
```

(a) Creating a Constraint Relation.

```
INSERT INTO T(name, submissions, groupId)
VALUES (
    'Michael Valdron',
    250,
    new_var()
);
```

(b) Insert a New Constraint Variable.

```
SELECT name,
       submissions,
       groupId? :>: 3 :and: groupId? :<: 7 as c
FROM T;
```

(c) Create Constraint from Relation.

Figure 1.1: SAT/SQL Examples.

Figure 1.1a. One can also create new tuples within a relation using the `INSERT INTO` statement from SQL which will create CP variables under VAR typed attributes, as shown in Figure 1.1b. Creating constraints from such a relation is also possible using the `SELECT` statement from SQL to create expressions that can contain some VAR typed attributes and extended operators which will form the constraints, as shown in Figure 1.1c. Extended operators in SAT/SQL are shown and discussed in Chapter 4.

Figure 1.1c also provides an example of using SAT/SQL to create a data representation of a constraint as mentioned in Section 1.2.1, further showing the use of Data-Driven Constraint Programming. SAT/SQL is the core of bring the relational DBMS and the SAT Solver together. More on SAT/SQL will be discussed further in detail in Chapter 4, with many case studies.

### 1.2.3 Iterative and Interactive Constraint Programming

With the modernization of programming languages comes environments such as the REPL. Various DBMSs even provide a REPL environment for performing SQL queries, such as PostgreSQL. Though the REPL environment can provide an iterative and interactive programming experience, there are still many REPL which need improving informativeness in error checking. The CP also have been lacking in this aspect in the iterative and interactive environment.

In our work, we provide improvements in informativeness within the area Iterative and Interactive Constraint Programming. These improvements come in two methodologies, Conflict Detection and Optimal Conflict Repair. Conflict Detection has some algorithm which takes a CP model as input and returns the identified minimal conflict constraint group found amongst the overall constraints in the model in the form of a set known as a *min-conflict set*. Optimal Conflict Repair has some algorithm which takes the same input as an algorithm for Conflict Detection, however, the return set is a set of constraints which if pruned from the overall constraints would fix a conflict in the model, this set is also known as the *min-repair set*. Both the concepts of the *min-conflict set* and *min-repair set* are discussed briefly in our other work [51].

With these methodologies, we provide a better iterative and interactive constraint programming experience in both SAT/SQL and other REPL environments. These methodologies will be discussed more in Chapter 5.

## 1.3 Contribution

Our work contributes to two areas of CS, namely CP and databases. For CP, we see additional work in data driven CP models including data driven SAT solvers. We also see the iterative and interactive environment improved with the ability to

compute the *min-conflict set* and *min-repair set* in real time. For databases, we bring a conceptual extension of SQL, SAT/SQL, and the idea of a DBMS which works with the additions brought by SAT/SQL.

The contributions also provide the benefit of CP integration into databases via the SAT/SQL extension language. With the iterative and interactive constraint programming features we present for improving the CP experience, we also bring that same benefit to SAT/SQL. As with standard SQL along with various DBMSs in today's market, SAT/SQL will also have a REPL environment along with iterative and interactive constraint programming improvements for the CP additions within the language.

## 1.4 Outline

We have provided a summary of what this literature will cover in greater detail in the upcoming chapters. This section will outline the upcoming chapters in chronological order. The outline is as follows:

- Chapter 2 provides background on CP concepts and mathematical logic.
- Chapter 3 goes into the background of relational data management and SQL.
- Chapter 4 defines SAT/SQL with detail and demonstrates SAT/SQL with case studies.
- Chapter 5 shows the methodologies for providing Iterative and Interactive coding to CP. The contents of this chapter include Conflict Detection, Conflict Repair, and a case study to demonstrate.
- Chapter 6 provides the implementation and evaluation details for the experimentation of the methodologies.

- Chapter 7 goes into detail on the related work in SAT Solvers. The related work's main focus is contributions in iterative and interactive programming with CP or SAT Solvers.

- Lastly, Chapter 8 summarizes the outcome of all research done and describes the directions we would like to see the work go in.

# Chapter 2

# Constraints & Mathematical Logic

Before one could get deep into the core content of the work done in this literature, we must look at some of the subject matter that makes up areas that this work utilizes. We shall begin looking into key points behind CP and SAT solvers.

In this chapter, we will present an overview of the background found in Constraints and Mathematical Logic. In the following sections, we present the preliminary concepts in **Propositional Logic**, **Integer Variables in CP**, and **Optimization**. Also, we will present examples of modelling using these concepts.

## 2.1 Propositional Logic

In this section, we visit the background of **Propositional Logic** and the use of this subject with NP-complete problems. The content in this literature assumes that the reader has knowledge in **Logic** and **Complexity Theory**. We recommend the literatures [4, 35, 46] to fill in any knowledge gaps these preliminary areas. We also refer to the literature [46] for the background in **Propositional Logic**.

**Definition 1** (Propositional Logic (PL))**.** *Naturally pure boolean and also referred to*

*as Propositional Calculus, expresses statements for problems within a boolean statement. As [46] defines it, logical rules which defines mathematical statements.*

One can apply PL in many situations. For example, we have a supervisor who organizes a meeting with an employee. Let us say we have a statement which states the following, *employee does not miss meeting with the supervisor.* The following mathematical expression shows this example in PL form:

$$S \implies E \tag{2.1}$$

Equation 2.1 shows PL representation of the statement, $S$ is the boolean which indicates if the supervisor shows up, and $E$ indicates if the employee shows up. If the supervisor does not show up, meaning $S$ is *false*, it matters not if the employee shows up as there is no meeting to miss. If the supervisor does show, meaning $S$ is *true*, the employee must show not to miss a meeting, meaning $E$ must be *true* for Equation 2.1 to be *true* in this case.

Though this a simple example, there are many more challenging problems in PL. We move on to two critical concepts for dealing with such problems.

### 2.1.1 Conjunctive & Disjunctive Normal Form

Most problems are more extensive and may contain multiple statements to define it. Thus, we need ways of connecting these mathematical statements.

$$p \wedge q \wedge r \wedge \ldots \tag{2.2}$$

Equation 2.2 shows a logical expression in a form referred to as a Conjunctive Normal Form (CNF). A CNF expression is *true* when all members of the expression hold *true*. Let us conciser a CNF with three members, $p \wedge q \wedge r$. If $p$, $q$, and $r$ are *true*

then in this case $p \wedge q \wedge r$ would also be *true*. Now lets say $r$ is *false*, this would mean the expression simplified to **True** $\wedge$ **True** $\wedge$ **False**, therefore the whole expression is now *false*.

$$p \vee q \vee r \vee \ldots \tag{2.3}$$

Now we have Equation 2.3, which shows an expression in a form referred to as a Disjunctive Normal Form (DNF). A DNF expression yields *true* if any of the members within the expression yield *true*. Let us conciser a DNF with three members, $p \vee q \vee r$. Now lets say $p$ and $q$ are *true* and $r$ is *false*, this would mean the expression simplified to **True** $\vee$ **True** $\vee$ **False**, therefore the whole expression is *true*. If all members are *false*, **False** $\vee$ **False** $\vee$ **False**, then the whole expression would be *false*.

### 2.1.2 Truth Tables

Truth Tables are a great way to break down PL expressions into subexpressions to verify which each of these evaluates for every outcome permutations. Each column in a Truth Table represents the permutations of outcomes for a component. Each row of a Truth Table shows the outcomes for every possible permutation of the expression components.

Table 2.1 shows an example the construction of a Truth Table for the expression $p \wedge (q \vee \neg r)$. Notice that the outcomes of variable components $p$, $q$, and $r$ match the number of the permutations of outcomes possible (number of rows). The permutations, in this case, $2^{V_n}$ where $V_n = 3$ variables, yields 9 permutations of outcomes, which allows us to create the rows and columns for the variable components which are shown in the first view of Table 2.1. Next, we fill an additional column(s) for single component(s) subexpressions. In this case, we only have one single component

| $p$ | $q$ | $r$ | $\neg r$ | $q \vee \neg r$ | $p \wedge (q \vee \neg r)$ |
|---|---|---|---|---|---|
| T | T | T | ? | ? | ? |
| T | T | F | ? | ? | ? |
| T | F | F | ? | ? | ? |
| T | F | T | ? | ? | ? |
| F | T | T | ? | ? | ? |
| F | F | T | ? | ? | ? |
| F | T | F | ? | ? | ? |
| F | F | F | ? | ? | ? |

| $p$ | $q$ | $r$ | $\neg r$ | $q \vee \neg r$ | $p \wedge (q \vee \neg r)$ |
|---|---|---|---|---|---|
| T | T | T | F | ? | ? |
| T | T | F | T | ? | ? |
| T | F | F | T | ? | ? |
| T | F | T | F | ? | ? |
| F | T | T | F | ? | ? |
| F | F | T | F | ? | ? |
| F | T | F | T | ? | ? |
| F | F | F | T | ? | ? |

| $p$ | $q$ | $r$ | $\neg r$ | $q \vee \neg r$ | $p \wedge (q \vee \neg r)$ |
|---|---|---|---|---|---|
| T | T | T | F | T | ? |
| T | T | F | T | T | ? |
| T | F | F | T | T | ? |
| T | F | T | F | F | ? |
| F | T | T | F | T | ? |
| F | F | T | F | F | ? |
| F | T | F | T | T | ? |
| F | F | F | T | T | ? |

| $p$ | $q$ | $r$ | $\neg r$ | $q \vee \neg r$ | $p \wedge (q \vee \neg r)$ |
|---|---|---|---|---|---|
| T | T | T | F | T | T |
| T | T | F | T | T | T |
| T | F | F | T | T | T |
| T | F | T | F | F | F |
| F | T | T | F | T | F |
| F | F | T | F | F | F |
| F | T | F | T | T | F |
| F | F | F | T | T | F |

Table 2.1: Truth Table construction of the logical expression $p \wedge (q \vee \neg r)$.

subexpression, $\neg r$, seen in the second view of Table 2.1. Then, we keep going up in subexpression size until we reach the entire expression, see the third view of Table 2.1 for $q \vee \neg r$. The last view of Table 2.1 has the shows the complete Truth Table with all broken down component outcomes with the addition of the full expression.

We revisit Equation 2.1 for the example statement in Section 2.1, we can produce the following Truth Table:

| $S$ | $E$ | $S \implies E$ |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

Even though we can use a simple expression such as Equation 2.1 for constructing a Truth Table, it is unnecessary as the permutations of outcomes for this expression would be relatively simple in itself to identify. Truth Tables are at their best when using expressions with multiple levels of components, such as shown in Table 2.1. More examples of Truth Tables can be found in [46].

## 2.2 Modelling using PL

In this section, we revisit the concepts of PL seen in Section 2.1 and in [46] to show how to model problems with PL. This section's content provides the core fundamentals one would need to enter the understanding of how basic CP models work. Throughout this section, we will look at a few complex problems to demonstrate using the concepts of PL to solve those said problems.

**Problem 1** (Scheduling)**.** *Three people are trying to schedule a meeting with each other. The individuals comprise Jack, Jill, and Joe. Jack is available to meet any-*

*where and any day except Wednesday. Jill is only available to meet in Toronto on Monday, Tuesday, and Wednesday, but can meet anywhere on Thursday and Friday. Joe is only available to meet in Oshawa on Monday and Friday. Where is a common meeting place, and when should they meet?*

In Problem 1, we have a scheduling problem between three individuals. We have common variables of the meeting, which could constraint each person from attending. These variables are a place $p$ and the day of week $d$. The possible values of days in the problem are $\mathrm{D} = \{M, T, W, R, F\}$ such that $d \in \mathrm{D}$. The possible locations in the problem are $\mathrm{P} = \{T, O\}$ such that $p \in \mathrm{P}$, $T$ as Toronto, and $O$ as Oshawa. Next, we need to model the statements which express the limits of each person. We first define a function $m(d, p)$, which gets a boolean that indicates if a meeting can happen on the day $d$ and at place $p$. Second, we express these statements in mathematical logic in the form of PL which will enforce our model's constraints.

$$C_1 = \bigwedge_{p \in \mathrm{P}} \neg m(W, p) \tag{2.4}$$

$$C_2 = \bigwedge_{d \in (\mathrm{D} - \{R, F\})} \neg m(d, O) \tag{2.5}$$

$$C_3 = \left( \bigwedge_{d \in (\mathrm{D} - \{M, F\})} \bigwedge_{p \in \mathrm{P}} \neg m(d, p) \right) \wedge \neg m(M, T) \wedge \neg m(F, T) \tag{2.6}$$

$$C = C_1 \wedge C_2 \wedge C_3 \tag{2.7}$$

The equations above show the overall model of constraints in the form of PL. Equation 2.4 expresses the restriction of Jack which cannot meet on Wednesdays. Equation 2.5 shows Jill's restriction, in which she can only meet in Toronto on some

days then can meet anywhere other days. Equation 2.6 shows the restriction of Joe, who can only meet in Oshawa on certain days. All restrictions are shown as constraints $C_1$, $C_2$, and $C_3$ respectively. Each constraint shows if the person the constraint represents can make it to a meeting.

Different from the other equations, Equation 2.7 shows the overall CNF for the model $C$. If $C$ is *true* then a meeting can happen whereas if $C$ is *false* then no meeting is possible. This is standard component of SAT solvers and any constraint which is added to a SAT solver model is then placed in a CNF expression.

For the next step in solving this problem, we turn to the use of Truth Tables. Since this problem is large, we will break the Truth Table for this model into parts, and not all the outcomes will be shown. Table 2.2 and Table 2.3 shows the Truth Table split into parts. The first three outcomes are shown along with an outcome $i$, which shows the only outcome with $C$ as *true*. From this Truth Table, we can see that outcome $i$ has all constraints as *true*. When looking at the tabular views which show the outcomes for the model boolean variables, we see that all variables except $m(F, O)$ are *false*. This outcome shows that the only day and place feasible for these individuals to meet is Friday and Oshawa, respectively.

## 2.3 PL with Integer Variables

In Section 2.2, we took a look at modelling a problem over PL using pure boolean expressions. We can model problems in PL with integers as well. In this section, we take a look at adding integers to the mix. Many problems in CP use integer variables either in pure integer problems or as an indexed representation for choices. Using integers in PL also can lead one to explore other forms of logic programming such as Linear Programming (LP) and Mixed Integer Programming (MIP).

| # | $m(M,T)$ | $m(T,T)$ | $m(W,T)$ | $m(R,T)$ | $m(F,T)$ |
|---|---|---|---|---|---|
| 1 | T | T | T | T | T |
| 2 | T | T | T | T | T |
| 3 | T | T | T | T | T |
| ... | ... | ... | ... | ... | ... |
| $i$ | F | F | F | F | F |
| ... | ... | ... | ... | ... | ... |

| # | $m(M,O)$ | $m(T,O)$ | $m(W,O)$ | $m(R,O)$ | $m(F,O)$ |
|---|---|---|---|---|---|
| 1 | T | T | T | T | T |
| 2 | T | T | T | T | F |
| 3 | T | T | T | F | F |
| ... | ... | ... | ... | ... | ... |
| $i$ | F | F | F | F | T |
| ... | ... | ... | ... | ... | ... |

| # | $\neg m(M,T)$ | $\neg m(T,T)$ | $\neg m(W,T)$ | $\neg m(R,T)$ | $\neg m(F,T)$ |
|---|---|---|---|---|---|
| 1 | F | F | F | F | F |
| 2 | F | F | F | F | F |
| 3 | F | F | F | F | F |
| ... | ... | ... | ... | ... | ... |
| $i$ | T | T | T | T | T |
| ... | ... | ... | ... | ... | ... |

Table 2.2: Truth Table (Part 1) for Problem 1.

| # | $\neg m(M,O)$ | $\neg m(T,O)$ | $\neg m(W,O)$ | $\neg m(R,O)$ | $\neg m(F,O)$ |
|---|---|---|---|---|---|
| 1 | F | F | F | F | F |
| 2 | F | F | F | F | T |
| 3 | F | F | F | T | T |
| ... | ... | ... | ... | ... | ... |
| $i$ | T | T | T | T | F |
| ... | ... | ... | ... | ... | ... |

| # | $\bigwedge_{d\in(\mathrm{D}-\{M,F\})}\bigwedge_{p\in\mathrm{P}}\neg m(d,p)$ |
|---|---|
| 1 | F |
| 2 | F |
| 3 | F |
| ... | ... |
| $i$ | T |
| ... | ... |

| # | $C_1$ | $C_2$ | $C_3$ | $C$ |
|---|---|---|---|---|
| 1 | F | F | F | F |
| 2 | F | F | F | F |
| 3 | F | F | F | F |
| ... | ... | ... | ... | ... |
| $i$ | T | T | T | T |
| ... | ... | ... | ... | ... |

Table 2.3: Truth Table (Part 2) for Problem 1.

Consider the following statements:

1. *Greg is older than Sally.*: Equation 2.8
2. *Greg is at most 25 years old.*: Equation 2.9
3. *Sally is at least 16 years old.*: Equation 2.10

$$g > s \tag{2.8}$$

$$g \leq 25 \tag{2.9}$$

$$s \geq 16 \tag{2.10}$$

With these statements, we consider two integer variables, Greg's age $g \in \mathbb{Z}$ and Sally's age $s \in \mathbb{Z}$. To satisfy the first statement translate to the PL expression shown in Equation 2.8. The next two statements define the bounds in the problem, referring to that Sally is at least 16, and Greg is at most 25. For this specific case, we could express these bounds in two different ways. The first method of expressing the bounds is to keep the definition of the variables to be members of $\mathbb{Z}$ space, therefore translate the statements to the PL expressions shown in Equation 2.9 and Equation 2.10. Another way of expressing these statements is to define the variables with a bounded range, such that $g \in [16, 25]$ and $s \in [16, 25]$. Since we know that no person in this problem can exceed these bounds the range $[16, 25]$ works, therefore Equation 2.9 and Equation 2.10 would be unnecessary. If Equation 2.8 did not exist however then the first method of expressing Equation 2.9 and Equation 2.10 would be the only valid one.

Now, what if we are dealing with a collection of integer numbers $X \subset \mathbb{Z}$? Let us say we have the following statements:

1. *All numbers in the collection $X$ must be different.*
2. *Some numbers in the collection $X$ can be even numbers.*
3. *Numbers in collection $X$ must total to* 100.

For these statements, we could use expressions with quantifiers as such:

$$\forall x \in X, \forall y \in X (x \neq y) \tag{2.11}$$

$$\exists x \in X (x \bmod 2 \equiv 0) \tag{2.12}$$

$$\sum_{x \in X} x = 100 \tag{2.13}$$

There is an issue with all three of these equations. Using quantifiers puts the expressions in first-order logic, which we cannot use in practical logic programming. We go more in detail with the issue of first-order logic in Section 2.4. For this example, we shall convert from our first-order logical expressions to expanded logical expressions, as shown below:

$$X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge \ldots \wedge X_1 \neq X_n \wedge X_2 \neq X_3 \wedge \ldots \wedge X_{n-1} \neq X_n \tag{2.14}$$

$$X_1 \bmod 2 \equiv 0 \vee X_2 \bmod 2 \equiv 0 \vee \ldots \vee X_n \bmod 2 \equiv 0 \tag{2.15}$$

```
  S E N D
+ M O R E
---------
M O N E Y
```

Figure 2.1: Send More Money Cryptarithmetic Problem

$$X_1 + X_2 + \ldots + X_n = 100 \tag{2.16}$$

**Problem 2** (Cryptarithmetics)**.** *A classical problem commonly used as show case CP [1]. We define a phrase then the resultant word. For this problem, let us use the one outlined by Nareyek [1] and shown in Figure 2.1. We start by defining variables for each letter in the problem; $S$, $E$, $N$, $D$, $M$, $O$, $R$, and $Y$. The letter to integer variables will make up a system of linear equations calculate the resultant number for the word. To solve the problem, assign integers to the letters such that the words making up the phrase sum to the resultant word, shown in Figure 2.1.*

Now lets look at Problem 2, which uses integers within PL. To start this problem, let $L$ be the set of letters $\{S, E, N, D, M, O, R, Y\}$ such that we can define an expression that ensures all letters are assigned different integers, such as we defined Equation 2.14. The expression is as follows:

$$L_1 \neq L_2 \wedge L_1 \neq L_3 \wedge \ldots \wedge L_1 \neq L_n \wedge L_2 \neq L_3 \wedge \ldots \wedge L_{n-1} \neq L_n \tag{2.17}$$

Equation 2.17 ensures all letter integers and the integer results of the words will not be the same as each other. Next, we define the expressions for each word:

$$w_1 = 1000S + 100E + 10N + D$$

$$w_2 = 1000M + 100O + 10R + E$$

$$w_3 = 10000M + 1000O + 100N + 10E + Y$$

Where $w_1$ is the integer variable for the word SEND, $w_2$ is the integer variable for the word MORE, and $w_3$ is the integer variable for the word MONEY. The expressions include variables for all letters and coefficients that start at 1 then scale up by a factor of 10 for each term. Notice that the same letters which share different words also share the same variable between the linear equations. This crossing of letters between words means the solution choice for a letter would need to satisfy every word in which that letter occurs.

These linear expressions will yield integer results for $w_1$, $w_2$, and $w_3$. We need to identify the phrase and the resultant word. $w_1$ and $w_2$ are part of the phrase. $w_3$ is the resultant word, which means that $w_3$ is the sum of the phrase. We will need one more expression for this:

$$w_1 + w_2 = w_3 \tag{2.18}$$

This last expression, Equation 2.18, enforces the sum on $w_1$ and $w_2$ to equal $w_3$. This also enforces the letters in common to find solutions which makes this equation work. This problem complex and falls under the NP-complete complexity space [21]. The good news is SAT solvers are perfect for solving problems within the NP-complete space. We ran this model on a SAT solver and got this solution for the letters:

$$\{S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2\}$$

Let us check if the letters solution from the SAT solver produces a feasible solution when we use them with our expressions:

$$\begin{aligned} w_1 &= 1000S + 100E + 10N + D \\ w_1 &= 1000(9) + 100(5) + 10(6) + (7) \\ w_1 &= 9567 \end{aligned}$$

$$\begin{aligned} w_2 &= 1000M + 100O + 10R + E \\ w_2 &= 1000(1) + 100(0) + 10(8) + (5) \\ w_2 &= 1085 \end{aligned}$$

$$\begin{aligned} w_3 &= 10000M + 1000O + 100N + 10E + Y \\ w_3 &= 10000(1) + 1000(0) + 100(6) + 10(5) + (2) \\ w_3 &= 10652 \end{aligned}$$

$$\begin{aligned} w_1 + w_2 &= w_3 \\ (9567) + (1085) &= (10652) \\ \mathbf{10652} &= \mathbf{10652} \end{aligned}$$

Indeed, subbing the solution we got into the expressions provides us with a pos-

sible answer. Adding integers to PL expressions give us a general understanding of modelling constraints in CP and SAT solvers.

## 2.4 First-Order Logic

Back in Section 2.3, we mentioned that Equation 2.11, Equation 2.12, and Equation 2.13 were in first-order logic which therefore could not be used in the form these expressions were in. In this section, we briefly discuss first-order logic and why there is an issue with using it when creating logical expressions such as PL expressions.

### 2.4.1 Predicates and Quantifiers

**Predicates** are functions with some operation to done on the arguments, which will have a boolean result:

$$P(x) \wedge Q(x)$$

**Quantifiers** perform an operation which relates to the quantity of something. This description is not very clear, so let us discuss some types and examples of quantifiers to shed more light on them. In mathematics, there are two types of quantifiers one can use in first-order logical expressions [46]:

- Universal Quantifiers '$\forall$'
- Existential Quantifiers '$\exists$'

**Universal Quantifiers** state that in a logical expression that all members of some domain hold true to the expression. For example, let $d$ be the representation of a member of some domain $D$:

$$\forall(d \in D)(d > 2) \tag{2.19}$$

Equation 2.19 shows that all $d$ from the domain $D$ must be greater than 2. **Existential Quantifiers** state that in a logical expression that some members of some domain hold true to the expression. For example, let us reuse the same members in Equation 2.19:

$$\exists(d \in D)(d > 2) \tag{2.20}$$

Equation 2.20 shows that some $d$ from the domain $D$ must be greater than 2. From Equation 2.19 and Equation 2.20, we indeed see that there is a semantic relationship imposed on every member of $d$.

### 2.4.2 Completeness and Undecidability

In 1929, a Ph.D. student and famous mathematician Kurt Gödel proved the **Gödel's Incompleteness Theorem** for his Ph.D. dissertation [30]. **Gödel's Incompleteness Theorem** proves that there is a relationship between the semantics and provability of the expressions within first-order logic.

So from **Gödel's Incompleteness Theorem**, we take away that first-order logic is only semidecidable where the first-order components of expression must have meaning to be complete. Without meaning, the first-order expressions are incomplete thus are undecidable, which is the issue posed by first-order logic.

In the later Chapter 4, we show how we can work with problems which would call for the use of first-order logical statements using our SQL language extension that solves this completeness and undecidability issue.

## 2.5 Optimization with Integer Programming

In this section, we now add Optimization to our PL modelling. We can find a solution using CP which is most optimal in accedence to the model given. Some CSPs need this for solving the problem entirely, meaning that a feasible solution is not enough.

Optimization in SAT solvers is done using an objective function. This objective function is actually an Integer Linear Programming (ILP) expression which constructs a puseudo-Boolean (PB) constraint. The PB constraint enforces the SAT solver to look for the solution that meets the objective of the PB constraint. There are two objectives one can invoke on ILP expressions as the objective function; *minimize* and *maximize*. When the objective is to *minimize*, the solver will push towards minimizing the resultant value of the ILP expression. When the objective is to *maximize*, the solver will push towards maximizing the resultant value of the ILP expression.

$$x_1 + x_2 + \ldots + x_n \tag{2.21}$$

Let $\phi$ be the objective function of a SAT solver. Expressing a *minimize* objective function can be done on the ILP expression shown in Equation 2.21:

$$\phi(\mathbf{x}) \leftarrow \min_{\mathbf{x} \in \mathbb{Z}^n} (x_1 + x_2 + \ldots + x_n) \tag{2.22}$$

Such that $\mathbf{x}$ is a vector in $\mathbb{Z}^n$ that contains every variable $x_i$, these are the inputs of the function. We can also express the *maximize* objective function similarly as follows:

$$\phi(\mathbf{x}) \leftarrow \max_{\mathbf{x} \in \mathbb{Z}^n} (x_1 + x_2 + \ldots + x_n) \tag{2.23}$$

The variable vector $\mathbf{x}$ in Equation 2.22 and Equation 2.23 is in $\mathbb{Z}^n$, however, we

can also have $\mathbf{x}$ be a vector of boolean variables in $\mathbb{B}^n$ or any vector space $A^n$ such that $A^n \subseteq \mathbb{Z}^n$. Let us use a CSP for to demonstrate objective functions in SAT solvers:

$$\mathbf{v} \leftarrow \langle 67, 150, 200, 160, 80, 25, 190, 14, 25 \rangle \tag{2.24}$$

$$\mathbf{w} \leftarrow \langle 10, 15, 50, 25, 5, 10, 5, 5, 20 \rangle \tag{2.25}$$

$$w_c \leftarrow 80 \tag{2.26}$$

**Problem 3** (Knapsack)**.** *We have a Knapsack which we need to fill. Each item to place into the Knapsack has a value and a weight. The Knapsack has a maximum capacity of weight which the total weight cannot exceed. The objective is to fill the Knapsack with enough items, which maximizes all the items' total value without exceeding total weight. Given the constants in Equation 2.24, Equation 2.25, and Equation 2.26 select the items* $\{\{v_0, w_0\}_0, \ldots, \{v_n, w_n\}_n\}$ *such that the total value* $x_1v_1 + x_2v_2 + \ldots + x_nv_n$ *is optimally max with the restriction of the total weight not exceeding the capacity* $x_1w_1 + x_2w_2 + \ldots + x_nw_n \leq w_c$*. For more details and history of the Knapsack Problem see survey by Assi [5].*

The Knapsack Problem shown as Problem 3 is a perfect small problem to model which requires an objective function $\phi$ to solve with a SAT solver. Let the model for Problem 3 look like the following:

**Constants:**

$$\mathbf{v}, \mathbf{w}, w_c \tag{2.27}$$

**Variables:**

$$\mathbf{x} : \mathbb{B}^n \tag{2.28}$$

**Objective:**

$$\phi(\mathbf{x}) \leftarrow \max_{\mathbf{x} \in \mathbb{Z}^n} (x_1 v_1 + x_2 v_2 + \ldots + x_n v_n) \tag{2.29}$$

**Constraints:**

$$c \leftarrow x_1 w_1 + x_2 w_2 + \ldots + x_n w_n \leq w_c \tag{2.30}$$

A variable $x_i$ within $\mathbf{x}$ defined in Equation 2.28 indicate if we choose item $i$ to be added to the Knapsack. The capacity restriction $w_c$ against the item weights $\mathbf{w}$ is defined in Equation 2.30 as $c$, which is the only constraint in the model. Our objective function $\phi(\mathbf{x})$ is defined Equation 2.29. $\phi(\mathbf{x})$ creates an objective for a solver to strive for, in this case maximizing the sum of item values $\mathbf{v}$. With an objective $\phi(\mathbf{x})$ and a constraint $c$ this model should now be able to provide an optimal solution to the Knapsack Problem.

Let us say we have this solution, denoted as $S_{\mathbf{x}}$, for the variables in $\mathbf{x}$:

$$S_{\mathbf{x}} \leftarrow \langle 1, 1, 0, 1, 1, 1, 1, 1, 0 \rangle \tag{2.31}$$

Let us check if $S_{\mathbf{x}}$ along with the constants from Equation 2.24, Equation 2.25, and Equation 2.26 produces a overall solution $S$ which is optimal:

$$x_1w_1 + x_2w_2 + \ldots + x_nw_n \leq w_c$$

$$(10) + (15) + (0)(50) + (25) + (5) + (10) + (5) + (5) + (0)(20) \leq 80$$

$$10 + 15 + 25 + 5 + 10 + 5 + 5 \leq 80$$

$$75 \leq 80$$

**TRUE**

It appears that our variable solution $S_{\mathbf{x}}$ holds *true* for the constraint. Calculating $\phi(\mathbf{x})$ with $S_{\mathbf{x}}$ and the same constants yields the following objective value $\phi_v(\mathbf{x})$:

$$\phi(\mathbf{x}) \leftarrow \max_{\mathbf{x} \in \mathbb{Z}^n}(x_1v_1 + x_2v_2 + \ldots + x_nv_n)$$

$$\phi_v(\mathbf{x}) \leftarrow x_1v_1 + x_2v_2 + \ldots + x_nv_n$$

$$\leftarrow (67) + (150) + (0)(200) + (160) + (80) + (25) + (190) + (14) + (0)(25)$$

$$\leftarrow 67 + 150 + 160 + 80 + 25 + 190 + 14$$

$$\leftarrow 686$$

We get with solution $S_{\mathbf{x}}$ an objective value $\phi_v(\mathbf{x})$ of 686 and a total weight $\sum_{i=0}^{n} w_i$ of 75. Well we have a feasible answer, but the question is $S_{\mathbf{x}}$ optimal? You will find that the answer is yes, if we were to try this on every other combination of values for the variables in $\mathbf{x}$ then we would either get infeasible answers or feasible non-optimal answers ($\phi_v(\mathbf{x}) < 686$). In SAT solvers, algorithms are used to find the optimal solution $S_x$ using the ILP expression $\phi(\mathbf{x})$ by the maximum value found in the search space.

This chapter has given an overview of the Constraints and Mathematical Logic preliminaries one would need to understand to read the chapters starting with Chapter 4. In Chapter 3, we discuss the relational data model and SQL preliminaries.

# Chapter 3

# Relational Data Management

Relational Database Systems are the most common databases in the industry, and relational data is the most common data model used along with objects. Relational Data Management is the job of modern relational DBMSs, providing structure and integrity to the relational data. In this chapter, we provide an overview of the Relational Data Model background and the standard way of interfacing with relational DBMSs, SQL. The fundamental knowledge in these topics will be essential when reading our literature starting with Chapter 4.

## 3.1 Relational Data Model

This section provides background on the Relational Data Model found in relational DBMSs and relational data files such as Comma Seperated Values (CSV) files. We recommend the text by Ramakrishnan et al. [44] for full in-depth knowledge into the Relation Model and Databases as a whole. We also recommend the text by Connolly et al. [16] for the relational DBMS background as it provides knowledge essential for academic and industrial use cases of relational DBMSs.

| | $a_1$ | $a_2$ | ... | $a_m$ |
|---|---|---|---|---|
| $t_1$ | $t_1[a_1]$ | $t_1[a_2]$ | ... | $t_1[a_m]$ |
| $t_2$ | $t_2[a_1]$ | $t_2[a_2]$ | ... | $t_2[a_m]$ |
| ... | ... | ... | ... | ... |
| $t_n$ | $t_n[a_1]$ | $t_n[a_1]$ | ... | $t_n[a_m]$ |

Table 3.1: Visual representation of a relation $T$.

There are many definitions given for the Relational Model (RM), but one which stands out is a definition given by Edgar F. Codd [11, 20], the first to propose the relational data model, who states that RM is a method of managing data in a format which is consistent with first-order predicate logic. Later in Chapter 4, we will revisit this definition as it poses a interesting challenge seen within Section 2.3 and discussed in Section 2.4.

Data within the RM is in the form of transactions, also known as tuples, which have columns which are attributes, a tabular format. Various DBMSs, datasets, and other data structure entities use the RM as the data model. In DBMSs, the RM is expanded to include groups of data structures. These data structures include *Relations* and *Views*. Let us discuss these RM groups.

A *Relation (or Table)* are the groups of the tabular data which the database stores. Relations contain the raw data, in other words, the data on the disk. Relational DBMSs are type strict, meaning attributes within relations have data types. Every DBMS has its data types and some common ones, such as INT or VARCHAR. For instance, PostgreSQL[1] would not have the exact same data types as Oracle[2], but will have some in common. Now let us go into the relational algebra[3] to describe the structure for relations.

Let us start with a relation $T$. $T$ contains tuples $T[t_1, t_2, \ldots, t_n]$ and attributes

[1]PostgreSQL Data Types
[2]Oracle Data Types
[3]See Chapter 4 of the text by Ramakrishnan [44] for background in relational algebra.

| | *id*: `INT` | *first_name*: `TEXT` | *last_name*: `TEXT` | *company*: `TEXT` |
|---|---|---|---|---|
| $t_1$ | 1023432 | Bob | Smith | That Awesome Business |
| $t_2$ | 1302443 | Michael | Valdron | Ontario Tech University |
| ... | ... | ... | ... | ... |
| $t_n$ | 1453212 | Sarah | Nickle | Organization Z |

Table 3.2: Visual representation of data in a relation named *Persons*.

$T(a_1, a_2, \ldots, a_m)$. Each attribute $a_j$ has a name denoted as $\text{name}_{attr}(a_j)$ and a data type $\tau_{a_j}$. Every tuple $t_i$ represents data entries within $T$. In every $t_i$ there is are cells for each $a_j$ denoted as $t_i[a_j]$ which contains data pertaining to the specifics of $a_j$ in $t_i$. Due to the type strict nature of Relational DBMSs $t_i[a_j]$ has to be restricted to the type $\tau_{a_j}$. We can see a visual representation of $T$ as Table 3.1.

Table 3.2 shows a relation named *Persons* with attributes *id*, *first_name*, *last_name*, and *company*. The tuples $Persons[t_1, t_2, \ldots, t_n]$ represents transactional records of persons in companies. Relations store the raw data records for which they represent, as such *Persons* would store all of these records within. There can be many tuples within relations and likely many attributes, which can be overwhelming for anyone or even computers to process anything out of these relations. Lucky for a long time, we have a solution to this.

A *view* (sometimes referred to as a Result Set) results from performing data aggregation against the tuples within a relation or relations. Views build at runtime, which shows the results of queries against databases. Views provide one with the ability to grab the data desired and aggregated data one wishes to produce from the raw records within relation(s). All these aspects of views allow one to remove the *visual* complexity typically found in relations and even be chained off themselves. The chaining of views can remove further *visual* complexity and preserve particular views with commonly used resultant data for reusability. Let us go into relational algebra to describe views and data aggregation.

| | $a'_1$ | $a'_2$ | ... | $a'_{m'}$ |
|---|---|---|---|---|
| $t'_1$ | $t'_1[a'_1]$ | $t'_1[a'_2]$ | ... | $t'_1[a'_{m'}]$ |
| $t'_2$ | $t'_2[a'_1]$ | $t'_2[a'_2]$ | ... | $t'_2[a'_{m'}]$ |
| ... | ... | ... | ... | ... |
| $t'_n$ | $t'_n[a'_1]$ | $t'_{n'}[a'_1]$ | ... | $t'_{n'}[a'_{m'}]$ |

Table 3.3: Visual representation of a view $V$ produced by a query $\mathrm{q}_c$.

Let $V$ represent a view, such that $V = T \bowtie_c T' \bowtie_c \ldots$ where $\bowtie_c$ is a conditioned join operation performed upon the relations $T$, $T'$, ... which can be one to many relations. It is important to specify that views are generated at runtime with a query $\mathrm{q}_c$, therefore we can also denote a view $V$ to be produced as $\mathrm{q}_c : (T \times T' \times \ldots) \mapsto V$ and denote an invoke as $\mathrm{q}_c(T \times T' \times \ldots) = V$. Just as relations, a view $V$ has $n'$ tuples $V[t'_1, t'_2, \ldots, t'_{n'}]$ and $m'$ attributes $V(a'_1, a'_2, \ldots, a'_{m'})$. We denote most of $V$'s members with a '′' suffix to differentiate from the members of relations. A visual representation of $V$ can be seen in Table 3.3.

Table 3.4 shows a view *OntarioTech* which shows everyone in relation *Persons*, seen in Table 3.2, that is associated with the *company* 'Ontario Tech University'. This view is produced by a *select* query which can be expressed using relational algebra:

$$\pi_{\mathbf{a}}(\sigma_c(Persons)) = OntarioTech$$

Where $c$ is the condition $company = \prime\text{Ontario Tech University}\prime$ and $\pi$ is the projection on attributes in **a** which only include *first_name*, *last_name*, and *company* from the relation *Persons*. Also, there only is one tuple $t'_1$ (or $t_2$ in *Persons*) which shows up in result set.

There are caveats with views, however. Since the view runs a query upon fetching view data, query optimization is essential when it is complicated. One method of optimization is to create an *index* for each relation. An *index* defines an attribute

| | *first_name*: `TEXT` | *last_name*: `TEXT` | *company*: `TEXT` |
|---|---|---|---|
| $t'_1$ | Michael | Valdron | Ontario Tech University |

Table 3.4: Visual representation of *OntarioTech* view.

or attributes whose values can quickly access their corresponding tuple. This kind of works like keys do in Hash Tables. It is best to choose an attribute or attributes that have unique values for uniquely identifying the tuples due to how an index works. Efficient use of these can significantly improve the performance of queries and view construction.

## 3.2 Structured Query Language

In the previous section, we talked about the standard RM and all the forms of data sources which utilizes it, such as relational DBMSs. In this section, we discuss the query language which interfaces with most relational DBMSs. The texts by Ramakrishnan [44] and Connolly [16] contain some information on SQL. We recommend the readings of Fontaine [24] for further background and applications in SQL. More education in SQL can be found in online resources[4].

Structured Query Language (SQL) is the standard language used by relational DBMSs for performing queries against the database. SQL was created by Donald Chamberlin and Raymond Boyce back in 1974 [9]. Based off of relational algebra and is considered a declarative language (and partial procedural) [44], SQL was proposed for using with Codd's relational model [11] within database Systems. Though Codd had argued that SQL deviates away from his original relational model principles [14], SQL has taken to be the standard language in use for relational database system in the industry [9].

[4]SQL background, examples and tutorials can be found at https://www.w3schools.com/sql/.

Relational DBMSs have their own slight variation of SQL statements, for these examples we will use PostgreSQL relational DBMS. Other DBMSs will have similar statements, see other sources which are specific to the relational DBMS of choice if not PostgreSQL.

Recall that we denote $\sigma_c$ as a select query with a $c$ condition. In relational algebra, we can this to express the following select statement:

$$\pi_{\mathbf{a}}(\sigma_c(T)) \bowtie \pi_{\mathbf{b}}(\sigma_d(R)) : \mathbf{a} \subseteq T(a_1, a_2, \ldots, a_m), \mathbf{b} \subseteq R(b_1, b_2, \ldots, b_{m'})$$

Where we have an overall condition $c$ on relation $T$, an overall condition $d$ on relation $R$, a selection of attributes $\mathbf{a}$ from the source relation $T$, a selection of attributes $\mathbf{b}$ from the source relation $R$, and join operator $\bowtie$. In SQL, this expression is translated into the block of source, which is known as a `SELECT` statement:

```
SELECT T.a1, T.a2, ..., R.b1, R.b2, ...
FROM T
NATURAL JOIN R
WHERE c AND d;
```

Notice that our join of the relations $T \bowtie R$ produced a `NATURAL JOIN` statement[5] within our `SELECT` statement. Let us focus on single relation `SELECT` statements for the remainder of this topic.

Let us revisit relation *Persons* shown in Table 3.2 and the view *OntarioTech* seen in Table 3.4 from Section 3.1. The view *OntarioTech* is created from a query expressed in relational algebra as:

$$\pi_{\mathit{first\_name},\mathit{last\_name},\mathit{company}}(\sigma_{\mathit{company}='\text{Ontario Tech University}'}(\mathit{Persons}))$$

[5]See Appendix B.5 for more details on `JOIN` statements.

We can express this equation using an SQL `SELECT` statement:

```
SELECT first_name, last_name, company
FROM Persons
WHERE company='Ontario Tech University';
```

This query will produce a tabular view similar to the view *OntarioTech*. It is important to note that this query does not create a view rather than the raw result set behind the view *OntarioTech*. We will revisit the rest of the query, which creates the view from the raw result set seen here. Now to continue, in SQL we can change the query to one that gets all tuples where *company* contains the substring 'University':

```
SELECT first_name, last_name, company
FROM Persons
WHERE company LIKE '%University%';
```

`CREATE` statements in SQL allows for us to create Databases, Schemas, Relations, and Views within a relational DBMS[6]. For creating a relation, let us revisit the relation named *Persons* shown in Table 3.2. For this relation, we have attributes with their data types in an ordered list below:

1. *id*: `INT`
2. *first_name*: `TEXT`
3. *last_name*: `TEXT`
4. *company*: `TEXT`

`CREATE` statements creating relations use the term table to refer to a relation by the keyword `TABLE`:

[6]See Appendix A.1 for database and schema creation details.

```
CREATE TABLE Persons(
    id INT,
    first_name TEXT,
    last_name TEXT,
    company TEXT
);
```

This `CREATE` statement creates an empty relation with the attributes with their data types enforced readily for appending tuples, which we shall get into later. `CREATE` statements creating views use SQL `SELECT` statements for fetching a result set of one or many relations:

```
CREATE VIEW OntarioTech AS (
    SELECT first_name, last_name, company
    FROM Persons
    WHERE company='Ontario Tech University'
);
```

As mentioned in Section 3.1, views are predefined but fetch the result set at runtime, so every time one queries a view, the SQL `SELECT` behind the view must run as well.

A `INSERT INTO` statement in SQL allows for us to create a tuple or tuples within a relation:

```
INSERT INTO Persons(id, first_name, last_name, company)
VALUES (
    1302443,
    'Michael',
    'Valdron',
```

```
        'Ontario Tech University'
    ), ...;
```

When using the `INSERT INTO` statement, the tuple to be inserted must follow the data types of the attribute(s) and the constraint(s) within the destination relation.

The `ALTER TABLE` statement allows for one to modify aspects of a relation. One can modify the attributes (columns) of some relation by adding new ones with `ADD`, deleting current ones with `DROP`, or changing attribute data types with an additional `ALTER`. In PostgreSQL, the `ALTER TABLE` statement is quite complex in how many relation modification actions can be performed, for more on `ALTER TABLE` visit the web resources[7] of the DBMS of choice. We have provided an example of changing the data type of an attribute:

```
ALTER TABLE Persons
ALTER first_name TYPE VARCHAR(150);
```

The `UPDATE` statement allows us to change values within tuples. When using update, it is also essential to use the `WHERE` clause to pick out a specific tuple. Otherwise, all tuples for the given attribute(s) will update:

```
UPDATE Persons
SET first_name='John', company='That Awesome Business'
WHERE id=1302443;
```

Similar to the `UPDATE` statement, the `DELETE` statement uses the `WHERE` clause to remove a specific tuple from the target relation:

```
DELETE FROM Persons
WHERE id=1023432;
```

If `WHERE` is not specified, `DELETE` will remove all tuples in the target relation.

[7]PostgreSQL: https://www.postgresql.org/docs/12/sql-altertable.html

# Chapter 4

# SAT/SQL: The Constrained Structured Query Language

In this chapter, we look at the first focus of our research, an extension of SQL which incorporates members of CP and SAT solving capability. With the SQL extension also includes the need to properly both express CP members as data and incorporate data into constraint declaration. The extension of SQL includes additional operators which creates CP variables and constraints. This extension is able to perform Create Read Update Delete (CRUD) operations against constraint relations and views.

Let us begin with a formal definition of the proposed SQL extension.

**Definition 2** (SAT/SQL)**.** *Given SQL, a query language for the use of interfacing with relational database system, we define an extension of this language which provides the ability of using Relational Constraints within relational DBMSs.*

SQL is highly expressive language for performing procedures on data within relational databases and with SAT/SQL one can retain this same expressiveness when performing complex problem solving with data-driven CSPs. It is also due to the popularity of SQL as a query language that we choose it for the the base of SAT/SQL.

| Type | Group | Origin |
|---|---|---|
| VAR | CP Entity | SAT/SQL |
| CONSTRAINT | CP Entity | SAT/SQL |
| FLOAT | $\mathbb{R}$ | SQL |
| INTEGER | $\mathbb{Z}$ | ... |
| TEXT | $G^1$ | ... |
| VARCHAR | $G^1$ | ... |
| TIMESTAMP | $G^1$ | ... |
| ENUM | $\langle \mathbb{Z}^+, G^1 \rangle$ | ... |
| ... | ... | ... |

Table 4.1: SAT/SQL Type List.

Even though we choose SQL, one could use this idea to achieve the similar results using different languages or even data model.

For SAT/SQL, we continue off our other work [51] which defines an extension of the RM to includes members of CP, called the *Constraint Relational Model (CRM)*. We also demonstrated the use of creating and using *constraint relations* (Definition 3) within the CRM [51].

**Definition 3** (Constraint Relation)**.** *A given relational database relation is a constraint relation if there is an attribute typed as a CP member.*

## 4.1 Including CP Members with Data

For CP members to exist within relations, encoding these members would need to be done. The resulting encoding for these CP members would lead to the extension of attribute types to include the following CP entity types:

- VAR[$\tau_{\text{data}}$]
- CONSTRAINT

[1] $G$ is Character Group

Where $\tau_{\text{data}} \subseteq \mathbb{Z}$ is the primitive data type of the CP variable's result. A attribute type $\tau$ for each attribute in a constraint relation can be any standard SQL types and the extended SAT/SQL types as shown in Table 4.1. With the extension of types, we can create constraint relations using the SAT/SQL:

```
CREATE TABLE T (
    a INTEGER,
    b TEXT,
    x? VAR[INTEGER]
);
```

As seen above, we can create a constraint relation $T$ which has columns $a$, $b$, and $x?$. Columns $a$ and $b$ have standard SQL types whereas column $x?$ has the extended type VAR[INTEGER] which creates a column which accepts CP variables in every tuple, each with a initial domain of $\mathbb{Z}$.

We now have our constraint relation, but we still need to populate it with tuples. In section 3.2 showed how we insert into a relation using SQL, in SAT/SQL this would be similar but now we have to create CP variables. We create variables using the `new_var` operator:

```
INSERT INTO T(a, b, x?)
VALUES (67, 'Michael', new_var());
```

As shown here, we use `new_var` operator as a constructor of sorts for creating the CP variable entity within the constraint relation. In the previous case, we know that the type of the attribute $x?$ is VAR[**INTEGER**] so `new_var` can be called without arguments, however, if the type of $x?$ is just VAR for instance then we would pass a type hint as such:

```
INSERT INTO T(a, b, x?)
VALUES (67, 'Michael', new_var(INTEGER));
```

The argument `INTEGER` would ensure that the CP variable created for this record has the domain of $\mathbb{Z}$. This operation can be done with any compatible $\tau_{\text{data}}$.

## 4.2 Constraint Operators

We can now create constraints to enforce $x?$ within our constraint relation $T$. In SAT/SQL we create constraints using an extension of *constraint operators* shown in Table 4.2. To separate the operators in SAT/SQL from those in standard SQL, we use a notation of `:operator:`, that is the operator surrounded by colons. To demonstrate these operators' use, let us say we want to specify a more restricted domain for $x?$ from $T$ to be between $[a, b]$. In mathematical logic, we can create the following expression for every instance of $x?$:

$$\forall x?(x? > a \wedge x? < b)$$

This equation, of course, uses first-order logic, which causes issues as described in Section 2.4. There is no way in mathematical logic to express this equation for relational data without resorting to first-order logic. SAT/SQL provides a solution for this by thinking in terms of relational algebra [44] with notation used in SQL. Therefore, we can express this using a `SELECT` statement:

```
SELECT (x? :>: a :and: x? :<: b)
FROM T;
```

With this SAT/SQL statement we produce a constraint entity for each record from $T$ which enforces an *and* constraint on two other constraint entities which enforce the

| Operator | Name | Description |
|---|---|---|
| `:=:` | Equality Operator | Creates an equality constraint on two literals such that lh = rh is enforced. |
| `:!=:` | Inequality Operator | Creates an inequality constraint on two literals such that lh $\neq$ rh is enforced. |
| `:>:` | Greater Than Operator | Creates a greater than constraint on two literals such that lh $>$ rh is enforced. |
| `:<:` | Less Than Operator | Creates a less than constraint on two literals such that lh $<$ rh is enforced. |
| `:>=:` | Greater or Equal To Operator | Creates a greater or equal to constraint on two literals such that lh $\geq$ rh is enforced. |
| `:<=:` | Less or Equal To Operator | Creates a less or equal to constraint on two literals such that lh $\leq$ rh is enforced. |
| `:not:` | Negate Operator | Creates a negation of a constraint $c$ such that $\neg c$ is enforced. |
| `:or:` | Or Operator | Creates a or constraint on two other constraints, $c_1$ and $c_2$, such that $c_1 \vee c_2$ is enforced. |
| `:and:` | And Operator | Creates an and constraint on two other constraints, $c_1$ and $c_2$, such that $c_1 \wedge c_2$ is enforced. |
| `:xor:` | Exclusive Or Operator | Creates an exclusive or constraint on two other constraints, $c_1$ and $c_2$, such that $c_1 \oplus c_2$ is enforced. |
| `:imply:` | Implementation Operator | Creates an implementation constraint on two other constraints, $c_1$ and $c_2$, such that $c_1 \implies c_2$ is enforced. |
| `:different:` | All Different Operator | Creates an all different constraint on a set of literals $L$ such that $\bigwedge_{i=0}^{\lvert L\rvert} l_i \neq l_{i+1}$. |

Table 4.2: SAT/SQL Constraint Operators.

| Operator | Name | Description |
|---|---|---|
| `:+:` | Addition Operator | Creates a literal that represents the result of an addition operation on multiple literals $l_1 + l_2 + \ldots$. |
| `:-:` | Subtraction Operator | Creates a literal that represents the result of a subtraction operation on multiple literals $l_1 - l_2 - \ldots$. |
| `:*:` | Multiplication Operator | Creates a literal that represents the result of a multiplication operation on two literals lh $\times$ rh. |
| `:/:` | Division Operator | Creates a literal that represents the result of a division operation on two literals $\frac{\text{lh}}{\text{rh}}$. |
| `:not:` | Negate Operator | Creates a negation of a literal $l$ such that $\neg l$ is the new literal. |

Table 4.3: SAT/SQL Expression Operators.

bounds $a$ and $b$ on every CP variable in the $x?$ attribute. SAT/SQL places the overall *and* constraint entity in the result set of this query. We can also create a view $V$ from this result set with `CREATE VIEW`:

```
CREATE VIEW V(c CONSTRAINT) AS
  (SELECT (x? :>: a :and: x? :<: b)
  FROM T);
```

This statement produces a tabular view with a single attribute $c$ typed CONSTRAINT to hold all the constraint entities retrieved in the result of the `SELECT` statement. These statements show we can use propositional logic in conjunction with SQL notation to perform CP on relational data. This results in form of an extension of relational algebra which we called *constraint relational algebra* [51].

## 4.3 Expression Operators

In CP, not all expressions are purely logical as well. For this we introduce *expression operators* shown in Table 4.3. These operators share the same notation as *constraint operator* (Table 4.2) but produce a literal entity for the result of the expression rather than a constraint entity. To start, we can use another constraint relation $R$ created from the following SAT/SQL statement:

```
CREATE TABLE R (
    x? VAR[INTEGER] ,
    y? VAR[INTEGER]
);
```

We can see the following expression in propositional logic:

$$4x? + 2y? \leq 0$$

We can express this using a SAT/SQL statement:

```
CREATE VIEW V(c CONSTRAINT) AS
    (SELECT ((4 :*: x?) :+: (2 :*: y?) :<=: 0)
    FROM R);
```

The `:<=:` operator creates our constraint to enforce but made of multiple expressions with expression operators. The terms `4 :*: x?` and `2 :*: y?` create two literals used in `term1 :+: term2` that also creates a literal for the constraint. These expression operators allow for the use of arithmetic to be done in constraint expressions within SAT/SQL while keeping to relational operations.

| Operator | Name | Description |
|---|---|---|
| `:sum:` | Summation Operator | Performs aggregate summation of all the literals in a passed attribute $a$ such that $\text{sum}(a) = \sum_{i=0}^{\lvert a \rvert} a_i$, then produces a literal for the result of the summation. |
| `:count:` | Count Operator | Performs aggregate count of all the literals in a passed attribute $a$ such that $\text{count}(a) = \lvert a \rvert$, then produces a literal for the result of the count. |
| `:every:` | Every Operator | Creates a constraint aggregation that enforces that all literals from the attribute hold true to the inner constraint entity $c$, as such $\text{every(c)} = \forall \text{a} : \text{c(a)}$. |
| `:some:` | Some Operator | Creates a constraint aggregation that enforces that some literals from the attribute hold true to the inner constraint entity $c$, as such $\text{some(c)} = \exists \text{a} : \text{c(a)}$. |
| `:distinct:` | Distinct Operator | Creates a constraint aggregation that enforces that all literal attributes will result in different solution values. |
| `:countdistinct:` | Count Distinct Operator | Creates a combination of the *Count Constraint* and the *Distinct Constraint*. |
| `:sumdistinct:` | Distinct Summation Operator | Creates a combination of the *Summation Constraint* and the *Distinct Constraint*. |

Table 4.4: SAT/SQL Aggregation Operators.

## 4.4 Aggregation Operators

Like in SQL, SAT/SQL can perform *aggregation operations* (Table 4.4) on constraint relations, both constraint aggregation and traditional aggregation. Consider the following scenario. We want to enforce that the summation of a particular attribute $x?$ in relation $T$ over all tuples $T[t_1, t_2, \ldots, t_n]$ is equal to a constant $a$. In a logical expression, this would yield to be first order logic:

$$a = \sum_{i=0}^{n} t_i[x?]$$

If we recall from earlier, we can create literals and constraints using extension operators within the *projection* of the `SELECT` statements. In SQL, we can use traditional aggregation operators within the *projection*:

```
SELECT SUM(A) AS sum_A
FROM T;
```

The same principle applies for SAT/SQL constraint aggregation operators (Table 4.4) as well:

```
SELECT (:sum:(x?) :=: a) AS c
FROM T;
```

We can also perform traditional SQL aggregation with SAT/SQL constraint aggregation:

```
SELECT organization,
       COUNT(*) AS row_count,
       (:sum:(x?) :=: a) AS c
FROM T
GROUP BY organization;
```

The above statement will group all records by `organization` attribute and get the number of records per organization, `row_count`. Along with the number of records, each organization will sum up the CP variables in $x?$ then enforce an equality constraint for each result with a constant $a$. This operation will create a constraint per organization.

## 4.5 Goal Statements

In the previous sections, we demonstrated how to build CP models using SAT/SQL operators. Of course, once we build our CP model in our data, we need to produce solutions from that model. In SAT/SQL, we need to explicitly create entities called *Goals* in order to create a solution space from our model.

**Definition 4** (Goal)**.** *An abstraction of constraints to form a common enforced objective. Such as with constraints, Goals are enforced and conditions must be met in order to be satisfiable. In SAT/SQL, this entity also adds model members to the underlying solver as well as creating a grouping for scoped solving.*

A DBMS using SAT/SQL would have a global registry, which stores all indices to variables which are stored in relations and views [51]. The DBMS would have a global variable scope $V$ which contains all variables created by queries, such as `CREATE TABLE` or `CREATE VIEW`:

$$V = \{v_1, v_2, \ldots, v_n\}$$

Such as with variables, Goals also have a global scope. The global goal scope $G$ stores the created Goals modelled as key-constraint pair:

$$k \mapsto \theta(x_1, x_2, \ldots, x_n)$$

Let $k$ be the unique key for the Goal, and $\theta$ be the constraint. The global goal scope is modelled with these key-constraint pairs:

$$G = \{k_i \mapsto \theta_i : i = 1 \ldots m\}$$

In SAT/SQL, we can add a Goal to $G$ with the `CREATE GOAL` statement:

**CREATE** GOAL *[name]* **AS** $\theta$

The *name* is the key for the goal. If we do not specify a *name* the DBMS will create a randomized string as the *name*. We can also create multiple Goals from constraints in `SELECT` statements:

**CREATE** GOALS **AS**

**SELECT** $a_1, a_2, \ldots, a_i, \theta$

**FROM** ...

Once we have the Goals created in $G$ we now have a complete CP program denoted as $\langle G, V \rangle$. In SAT/SQL, we solve for a solution with the `SOLUTION` operator in a `SELECT` statement:

**SELECT** x?, y?, ...

**FROM** SOLUTION(R) ...;

Where $x?, y?, \ldots$ are the variable attributes to solve, and $R$ is the parent constraint relation. When `SOLUTION(R)` is executed in the above `SELECT` statement, attributes $x?, y?, \ldots$ in the statements result set is the produced solution for $x?, y?, \ldots$ in $R$.

| city | state |
|---|---|
| San Francisco | CA |
| Los Angeles | CA |
| Las Vegas | NV |
| Salt Lake City | UT |
| ... | ... |

(a) *Cities*.

| city | activity |
|---|---|
| San Francisco | Shopping |
| San Francisco | Hiking |
| Los Angeles | Shopping |
| Los Angeles | Restaurant |
| Las Vegas | Shopping |
| Las Vegas | Theatre |
| Salt Lake City | Hiking |
| ... | ... |

(b) *CityActivities*.

Table 4.5: Travel Planning Relations.

## 4.6 Case Studies

Now we will dive into some case studies to demonstrate the use and advantages of using SAT/SQL. Our case studies comprise of CSPs of our design to best match real-world problems. We will now go into our case studies and CSPs *Travel Planning* and *Segment Coloring*.

### 4.6.1 Travel Planning

The first case study we looked at in our other work [51], which solves a CSP called *Travel Planning*. We redefine the *Travel Planning* CSP in Problem 4. We shall go over the steps to solve the CSP described in Problem 4 and build up a model for it using SAT/SQL statements.

**Problem 4** (Travel Planning)**.** *We want to plan a road trip across the states in the United States. We have created a few relations in Table 4.5,* `Cities` *shown in Table 4.5a lists the cities with their states and* `CityActivities` *shown in Table 4.5b lists the city-activity pairs. We want to plan the road trip under the following criteria:*

1. *The trip lasts three days, and we need to visit a different city per day.*

| day | city? |
|---|---|
| 1 | ? |
| 2 | ? |
| 3 | ? |

(a) *TravelPlan*.

| day | time | act? |
|---|---|---|
| 1 | morning | ? |
| 1 | afternoon | ? |
| 2 | morning | ? |
| 2 | afternoon | ? |
| 3 | morning | ? |
| 3 | afternoon | ? |

(b) *ActivityPlan*.

Table 4.6: Travel Planning Constraint Relations.

2. *We need to visit at least two states on our trip.*

3. *We need to do two activities per day, one in the morning and one in the afternoon.*

4. *We want to at least experience five different ones for the duration of the trip.*

We have the two relations `Cities` and `CityActivities` shown in Table 4.5 that shows the travel information for our travel plan. Now we need the relations for the travel plan itself. In our travel plan, we have two types of choices: the city for the day and the two activities for the city. To solve with SAT we would create CP variables for each of those choices. We can use constraint relations to achieve this using SAT/SQL. Our first constraint relation is `TravelPlan` (Table 4.6a), which assigns a city to a day:

```
CREATE TABLE TravelPlan (
    day INTEGER,
    city? VAR[INTEGER]
);
```

Our second constraint relation is `ActivityPlan` (Table 4.6b), which assigns an activity to a day and time of day:

```
CREATE TABLE ActivityPlan (
```

```
    day INTEGER,
    time ENUM('morning', 'afternoon'),
    act? VAR[INTEGER]
);
```

Now with our constraint relations created, we start creating our constraints mentioned in Problem 4. The first constraint that we shall denote as $c_1$ enforces the rule that all the cities in attribute `city?` from `TravelPlan`. In SAT/SQL, $c_1$ can be created by using the `:distinct:` aggregation operator (Table 4.4) in a `SELECT`. For our constraints to be added to the SAT solver program $\langle G, V \rangle$ within the DBMS, we need to express our constraints as goals. We can create a goal for $c_1$ by using the following statement:

```
CREATE GOAL AS
    SELECT :distinct:(city?) AS c1
    FROM TravelPlan;
```

The second constraint that we shall denote as $c_2$ enforces the rule that we need to visit at least two different states. This constraint is more complex and will need a view, shown in Table 4.7a, that connects our `city?` variables to a collection of new variables for the states, `state?`:

```
CREATE VIEW TravelPlanStates AS
    SELECT day, city?, new_var() AS state?
    FROM TravelPlan;
```

The view attribute `state?` will have the CP variable type `VAR[INTEGER]`. This view will connect the selected city value in `city?` to a state value in `state?`. To enforce the selected city to match the proper state, we need to join our view `TravelPlanStates` to the relation `Cities` shown in Table 4.5a:

```
CREATE GOALS AS
    SELECT Cities.name AS city, Cities.state AS state
        (city? :=: city :=>: state? :=: state) AS cs
    FROM TravelPlanStates
    NATURAL JOIN Cities;
```

These additional goals are created from an addition constraint $c_s$ that enforces valid city to state relationships in solutions. Now we can create a goal for $c_2$ with the following statement:

```
CREATE GOAL AS
    SELECT :countdistinct:(state?) :>=: 2 AS c2
    FROM TravelPlanStates;
```

The third constraint that we shall denote as $c_3$ enforces the rule that we need an activity in the morning and another in the afternoon. Like with $c_2$, $c_3$ is complex and will need another view, shown in Table 4.7b, that separates morning and afternoon activities as their own attributes `act_morng?` and `act_aftn?` respectively from constraint relation `ActivityPlan`:

```
CREATE VIEW ActivityPlan2 AS
    SELECT day,
            M.act? AS act_morng?,
            A.act? AS act_aftn?
    FROM ActivityPlan M
    NATURAL JOIN ActivityPlan A
    WHERE M.time = 'morning'
        AND A.time = 'afternoon';
```

| day | city? | state? |
|---|---|---|
| 1 | ? | ? |
| 2 | ? | ? |
| 3 | ? | ? |

(a) *TravelPlanStates.*

| day | act_morng? | act_aftn? |
|---|---|---|
| 1 | ? | ? |
| 2 | ? | ? |
| 3 | ? | ? |

(b) *ActivityPlan2.*

Table 4.7: Travel Planning Views.

With the view `ActivityPlan2`, we now can create goals for $c_3$ with the following statement:

```
CREATE GOALS AS
    SELECT act_morng? :!=: act_aftn? AS c3
    FROM ActivityPlan2;
```

The fourth constraint that we shall denote as $c_4$ enforces the rule that we need to do at least five different activities for our entire trip. This constraint will have a similar expression to $c_2$ but can be enforced directly on constraint relation `ActivityPlan`. We can create a goal from $c_4$ with the following statement:

```
CREATE GOAL AS
    SELECT :countdistinct:(act?) :>=: 5 AS c4
    FROM ActivityPlan;
```

Now we have created goals for all four rules $c_1, c_2, c_3, c_4$ it appears that we have completed our CP model. However, there is still one more hidden rule we have not yet accounted for, and it involves the relation `CityActivities` (Table 4.5b), which we have not used yet. We need to enforce that the chosen activity must be available in the chosen city for the day slot. We can create goals for this using the following SAT/SQL statement:

```
CREATE GOALS AS
    SELECT CityActivities.city, CityActivities.act
```

```
        :some:(city? :=: city :=>: act? :=: act) AS c_ca
    FROM TravelPlanStates
    NATURAL JOIN Cities;
```

We denote this overall constraint as $c_{ca}$ that will ensure all activities chosen are available for all the cities chosen in each day slot. Now with the CP model complete for the CP program $\langle G, V \rangle$ we can now solve for a solution for each constraint relation. We can use the following SAT/SQL statement for getting a solution for `TravelPlan`:

```
SELECT * FROM SOLUTION(TravelPlan);
```

We can use a similar SAT/SQL statement for getting a solution for `ActivityPlan`:

```
SELECT * FROM SOLUTION(ActivityPlan);
```

Since `TravelPlan` and `ActivityPlan` constraint relations are connected in $\langle G, V \rangle$, the first use of the `SOLUTION` would solve the solution space for all $\langle G, V \rangle$, which also includes `ActivityPlan`. This means the second use of the operator `SOLUTION` will only fetch the solution values for the variables in `ActivityPlan` rather than solve them again.

The result sets of each of these queries would be a bit cumbersome to read. Therefore, the following statement could better the viewing experience:

```
CREATE VIEW TravelPlanSolution AS
    SELECT
        T.day,
        T.city?
        M.act? AS act_morng?,
        A.act? AS act_aftn?
    FROM SOLUTION(TravelPlan) T
    NATURAL JOIN SOLUTION(ActivityPlan) M
```

| day | city? | act_morng? | act_aftn? |
|---|---|---|---|
| 1 | San Francisco | Shopping | Hiking |
| 2 | Los Angeles | Restaurant | Shopping |
| 3 | Las Vegas | Theatre | Casino |

Table 4.8: *TravelPlanSolution* View.

```
NATURAL JOIN SOLUTION(ActivityPlan) A
WHERE M.time = 'morning'
    AND A.time = 'afternoon';
```

Table 4.8 shows a valid result that the above view might produce. Therefore, this study shows that we can solve this CSP with data using SAT/SQL.

### 4.6.2 Segment Coloring

For the second case study, we shall look at a solution to the CSP *Segment Coloring*, which is defined in Problem 5. We shall go over the steps to solve the CSP described in Problem 5 and build up a model for it using SAT/SQL statements.

**Problem 5** (Segment Coloring)**.** *We have a collection of segments $S$. Each segment $s$ will be assigned a position which does not overlap each other and a color $s^c$. The colors to assign are finite and include* ***red****,* ***blue****,* ***yellow****, and* ***green****. If $|S|$ exceeds or is the same number of colors available to assign, then all colors must be used on all segments. Segments adjacent to each other must not have the same color, Equation 4.2.*

We shall start by creating the constraint relations, shown in Table 4.9, for our segments. Our first constraint relation `Segments`, shown in Table 4.9a, contains all the base segment data such as the bounds, length, and segment identifier:

```
CREATE TABLE Segments (
```

| id | start? | length | end? |
|---|---|---|---|
| 1 | ? | 23 | ? |
| 2 | ? | 44 | ? |
| 3 | ? | 12 | ? |
| ... | ... | ... | ... |

(a) *Segments.*

| id | color? |
|---|---|
| 1 | ? |
| 2 | ? |
| 3 | ? |
| ... | ... |

(b) *SegmentColoring.*

Table 4.9: Segment Coloring Constraint Relations.

```
    id INTEGER,
    start? VAR[INTEGER],
    length INTEGER,
    end? VAR[INTEGER]
);
```

Our second constraint relation `SegmentColoring`, shown in Table 4.9b, contains the CP variables for assigning the colors:

```
CREATE TABLE SegmentColoring (
    id INTEGER,
    color? VAR[enum('red', 'blue', 'yellow', 'green')]
);
```

With our constraint relations, we can start to enforce the constraints and goals. We first need to ensure that all of the bounds in our segments, which are attributes `start?` and `end?` follow the rules for segments. These rules will be our first goal $g_1$. There two of these rules, which are:

1. The length of segment $s^l$ must be consistent with the *start* bound $s^s$ and the *end* bound $s^e$: $s^l = s^e - s^s$.

2. The length of segment $s^l$ must be a positive integer: $s^l \geq 0$.

We can let $c_1$ be the constraint expression $s^l = s^e - s^s$ and $c_2$ be the constraint expression $s^l \geq 0$. Both of $c_1$ and $c_2$ make up $g_1$. In SAT/SQL, we express $g_1$ to enforce these constraints on all segment records:

```
CREATE GOALS AS (
    SELECT length :=: end? :-: start? AS c1,
           length :>=: 0 AS c2
    FROM Segments
);
```

Second, we need to address no overlapping segments, as mentioned in Problem 5. We can address no overlapping segments with the following:

$$c_3 \leftarrow s_i^e \leq s_j^s \vee s_j^e \leq s_i^s \tag{4.1}$$

Where $i$ and $j$ belong to the combination of segment index pairs $\binom{|S|}{2}$. We can enforce the constraint in Equation 4.1 to the segments as our second goal $g_2$ using SAT/SQL:

```
CREATE GOALS AS (
    SELECT (S1.end? :<=: S2.start?
            :or: S2.end? :<=: S1.start?) AS c3
    FROM Segments S1, Segments S2
    WHERE S1.id < S2.id
);
```

The above SAT/SQL performs $S \times S$ with the additional `WHERE` clause that applies that order does *not* matter for $\binom{|S|}{2}$. Third, we need to address the first coloring assignment goal $g_3$ that enforces different colors for adjacent segments. The constraint expression $c_4$ for $g_3$ is shown below:

$$c_4 \leftarrow \text{adj}(s_i, s_j) \implies s_i^c \neq s_j^c \tag{4.2}$$

Where $i$ and $j$ are the segment indices for $S \times S$. The predict $\text{adj}(s_i, s_j)$ results as **true** if the two segments $s_i$ and $s_j$ are not adjacent to each other. In this case, segments are adjacent to each other if $s_i$ and $s_j$ side by side:

$$\text{adj}(s_i, s_j) \leftarrow s_i^e = s_j^s \tag{4.3}$$

With Equation 4.3, we can simplify Equation 4.2 down to:

$$c_4 \leftarrow s_i^e = s_j^s \implies s_i^c \neq s_j^c \tag{4.4}$$

Equation 4.4 shows the $c_4$ expression with the 1D function for $\text{adj}(s_i, s_j)$ (Equation 4.3). We can apply $c_4$ as $g_3$ in a SAT/SQL statement:

```
CREATE GOALS AS (
    SELECT (S1.end? :=: S2.start?
            :=>: SC1.color? :!=: SC2.color?) AS c4
    FROM (Segments S1
          LEFT JOIN SegmentColoring SC1
          ON S1.id = SC1.id),
         (Segments S2
          LEFT JOIN SegmentColoring SC2
          ON S2.id = SC2.id)
);
```

Because we store our segment coloring information in the separate constraint relation `SegmentColoring` (Table 4.9b), we use `LEFT JOIN` clauses to connect the

segment coloring variables to the base segment records in `Segments` (Table 4.9a). We can consider the primary key attribute `id` in `SegmentColoring` to be a foreign key that points to `id` in `Segments`.

Fourth, we need to enforce our final goal $g_4$, which ensures the utilization of all colors if the segments match the total colors available. To start, let $S_c$ be the set of coloring variables for each segment, such that $s^c \in S_c$. Let distinct be a function that returns the variable results with distinct values:

$$\text{distinct} : S_c \mapsto S'_c$$

Where $S'_c$ is the set of variable results from $S_c$ with distinct values. We can use distinct($S_c$) with the constraint expression $c_5$ for $g_4$:

$$c_5 \leftarrow (|S| = n_{\text{colors}}) \implies (|\text{distinct}(S_c)| \geq n_{\text{colors}}) \tag{4.5}$$

Where $n_{\text{colors}}$ is the number of color assignments. The expression $c_5$ can be expressed in $g_4$ with a SAT/SQL statement:

```
CREATE GOAL AS (
    SELECT
        COUNT(*) AS NumSegs,
        (NumSegs :>=: 3
         :=>: :countdistinct:(color?) :>=: 3) AS c5
    FROM SegmentColoring
);
```

In this case, $n_{\text{colors}} = 3$. The constraint relation `SegmentColoring` will have $|S|$ records, therefore no `JOIN` with `Segments` is needed.

Now that we have all our goals created, we can now get the solution to our

| id | start? | length | end? | color? |
|---|---|---|---|---|
| 1 | 0 | 23 | 23 | red |
| 2 | 23 | 44 | 67 | blue |
| 3 | 67 | 12 | 79 | yellow |
| 4 | 79 | 4 | 83 | green |
| 5 | 83 | 7 | 90 | red |
| ... | ... | ... | ... | ... |

Table 4.10: Segment Coloring Solution View.

constraint relations with the following SAT/SQL statement, which will produce the result set shown in Table 4.10:

```
SELECT S.id, S.start?, S.length, S.end?, SC.color?
FROM SOLUTION(Segments) S
NATURAL JOIN SOLUTION(SegmentColoring) SC
ORDER BY S.start?;
```

# Chapter 5

# Iterative and Interactive Constraint Programming with SAT/SQL

In the previous chapter, we have seen that we can build models for CSPs within relational databases using *constraint relational algebra*. This ability creates the possibility of better support for scalable optimization problems in a real-time application.

Of course, with better scalability comes the potential of infeasible CP models. As constraints scale up, the more likely it is to run into conflicting constraints within the model, thus creating infeasibility. Same as with finding problems within code, debugging will to find conflicts within CP model. We propose operators that can perform such a debugging procedure to better support iterative and interactive CP. The underlying operations for these operators would utilize the underlying solver used to solve the CSP the operation targets.

Referring back to Chapter 4, a relational constraint program $\langle G, V \rangle$ will have a collection of goals $(k, \theta) \in G$ which add a constraint enforcement over the variables

$V$, where $k$ is the unique identifier for the constraint $\theta$. As with all CP models, these cumulates to a CNF expression $\bigwedge_{(k,\theta)\in G} \theta$.

As we mention in our previous work [51], there are several reasons which a relational constraint program $\langle G, V \rangle$ could be infeasible:

- Error(s) in the relational constraint queries
- Inherit conflicts in the data
- Unreasonable expectations of the goals

One could use our methods for finding these such problems with infeasibility as one would find problems with any iterative and interactive debugging method.

In the upcoming sections, we will first define *Goal Types* which will differentiate goals for when enforcement of constraints in CP models is not mandatory. Then lastly, we will define *conflict sets*, *repair sets*, and a *max-sat* solution method for assisting with debugging CP programs as well as relational constraint programs such as mentioned in Chapter 4. Within these sections, we will also define the relationships between all of these concepts.

## 5.1 Goal Types

In many SAT solvers, such as CP-SAT [42], constraints have the property of being enforceable or optional. For the goals in the upcoming sections, we will follow a similar pattern with two types of goals:

- required goals: $G_{\text{req}}$
- optional goals: $G_{\text{opt}}$

Required goals $G_{\text{req}}$ and optional goals $G_{\text{opt}}$ are collections which cumulate to the overall collection of goals $G$ within the CNF, such that $G = G_{\text{req}} \cup G_{\text{opt}}$ and $G_{\text{req}} \cap G_{\text{opt}} = \emptyset$. In this case, $\langle G_{\text{req}}, V \rangle$ needs to be feasible in order to get a solution whereas $\langle G, V \rangle$ could be feasible but not necessarily.

## 5.2 MIN-CONFLICT Sets

In the case of an infeasible constraint program $G$, our first objective is to localize the source of conflict. This objective leads to the first concept *conflicting sets* and *minimal conflicting sets.*

**Definition 5** (Conflicting Sets)**.** *Let $C \subseteq G$ be subset of the goals in $G$, that contains the required goals $G_{\text{req}} \subseteq C$. We say that $C$ is conflicting, or that $C$ is a conflicting set if $(C, V)$ is* on solution*. Furthermore, the set $C$ is* minimally conflicting*, or a* minimal conflicting set *if no strict subsets of $C$ are conflicting.*

*We will denote $\mathcal{C}$ be the set of* all minimal conflicting sets*.*

**Example 1.** *Consider the following $G$:*

| *type* | *key* | *goal* |
|---|---|---|
| *required* | $g_0$ | $x_1 + x_2 + x_3 + x_4 \leq 10$ |
| *optional* | $g_1$ | $x_1 \geq 7$ |
| *optional* | $g_2$ | $x_2 \geq 4$ |
| *optional* | $g_3$ | $x_3 \geq 3$ |
| *optional* | $g_4$ | $x_4 \geq 3$ |

- $C_0 = \{g_0, g_1, g_2, g_3, g_4\}$ *is conflicting, but it's not minimal.*
- $C_1 = \{g_0, g_1, g_3, g_4\} \subset C_0$ *is minimally conflicting since no subset.*

- *Minimally conflicting sets are not unique. The conflicting set with the fewest optional goals is* $C_3 = \{g_0, g_1, g_2\}$.

**Fact 1.** *Finding a conflicting set is NP-complete.*

In case that $G_{\text{req}}$ is feasible, but $G$ is infeasible, it is helpful to understand the source of the conflicts in $G_{\text{opt}}$. Finding a minimal conflicting set $C^*$ can be very helpful to the user. We introduce the conflict detection operator:

$$\text{con} : G \mapsto C^*$$

Where $G_{\text{req}} \subseteq C^* \subseteq G$ is a minimally conflicting set. Following the NP-completeness of satisfiability [26], finding any conflicting set is NP-complete, and so is finding a minimal conflicting set. Fortunately, we can utilize existing efficient solvers to implement the con operator, as shown later.

## 5.3 MIN-REPAIR Sets

Another way to assist the user in resolving conflicts is to automatically compute *repairs* to an infeasible problem.

**Definition 6** (Repair sets)**.** *Let* $R$ *be a set of goals such that* $R \subseteq G_{\text{opt}}$*. We say that* $R$ requires $G$*, or that* $R$ *is a* repair set *if* $G - R$ *becomes feasible. If, furthermore, no strict subsets of* $R$ *repairs* $G$*, then we say that* $R$ *minimally repairs* $G$*, or that* $R$ *is a minimal repair set of* $G$*.*

From Definition 6, we define the minimal repair operator as:

$$\text{require} : G \mapsto R^*$$

Where $R^*$ is a minimal repair of $G$. The repair sets are optional goals where removal from $G$ will make the overall program $\langle G, V \rangle$ feasible.

**Example 2.** *Continue with Example 1. Observe:*

- $R_0 = \{g_2, g_3, g_4\}$ *repairs* $G$*, but it's not minimal.*
- $R_1 = \{g_2, g_3\}$ *is a minimal repair.*
- $R_2 = \{g_1\}$ *is a minimal repair with the fewest optional goals.*

**Fact 2.** *Finding repair sets is NP-complete.*

Let us look at the relations between conflicting sets and repair sets. Recall that $\mathcal{C}$ is the set of all minimal conflicting sets.

**Theorem 1.** $\bigcup \mathcal{C}$ *is a repair set.*

To prove Theorem 1, we establish a more straightforward claim.

**Proposition 1.** *Any subset of goals,* $X \subseteq G$ *such that* $X \cap \bigcup \mathcal{C} = \emptyset$ *must be satisfiable.*

*Proof.* We can prove by contradiction. If $X$ is not satisfiable, then $X$ must be a conflicting set, and hence must overlap with some minimal conflicting sets in $\mathcal{C}$, which leads to a contradiction to the assumption that $X \cap \bigcup \mathcal{C} = \emptyset$. □

*Proof of Theorem 1.* Using Proposition 1, we can conclude that:

$$\left(G - \bigcup \mathcal{C}\right) \text{ is satisfiable}$$

By the definition of repair sets, we see that $\bigcup \mathcal{C}$ is a repair set. □

We should remark that $\bigcup \mathcal{C}$ is not necessarily minimal.

**Example 3.** *We can demonstrate in Example 1 that the minimal conflicting sets are:*

$$\mathcal{C} = \{\{g_1, g_2\}, \{g_1, g_3, g_4\}\}$$

*So, the repair we get is:*

$$\bigcup \mathcal{C} = \{g_1, g_2, g_3, g_4\}$$

**Corollary 1.** *Every minimal repair set $R$ is such that:*

$$R \cap \bigcup \mathcal{C} \neq \emptyset$$

The corollary relates minimal repairs to the conflicting sets. We will be utilizing this relation to construct efficient algorithms to compute minimal conflicting sets.

## 5.4 The MAX-SAT Solution

A MAX-SAT problem is a constraint satisfaction problem $(G, V)$ with an additional objective function over the variables:

$$h : \text{solution}(V) \rightarrow \mathbb{R}$$

So, a MAX-SAT problem is given as $(G, V, h)$. A solution to the MAX-SAT problem is an assignment of variables $V$ such that $h$(assignment) is maximized while satisfying the goals in $G$.

While MAX-SAT is also NP-complete, recent advances in optimization and constraint satisfaction libraries [42] has made MAX-SAT very efficient and practical for many real-life scale problems. Using MAX-SAT we can find a minimal repair set.

- For each $\theta_i \in G_{\text{opt}}$, we create a fresh 0/1 variable $b_i$.

- For each $\theta_i \in G_{\text{opt}}$, we construct a new required goal:

$$\phi_i = ((b_i = 1) \implies \theta_i)$$

- Construct a new set of goals:

$$G' = G_{\text{req}} \cup \{\phi_i\} \quad V' = V \cup \{b_i\}$$

- Construct an objective function:

$$h = \text{maximize}(\sum_i b_i)$$

**Theorem 2.** *Given a solution to* $\mathbf{MAXSAT}(G', V', h)$*:*

$$\text{assignment} : V' \to \text{values}$$

*The set*

$$\{\theta_i : b_i = 0\} \subseteq G_{\text{opt}}$$

*is a minimal repair.*

Thus, Theorem 2 provides an algorithmic way to implement the minimal repair operator:

**Algorithm 1: repair**$(G, V)$

**Input:** Goals $G$, Variables $V$
**let** $G' = G$
**let** $V' = V$
**for** $i \leftarrow 0$ **to** $|G_{\text{opt}}|$ **do**
    $b_i = \text{new\_var}()$
    $\phi_i = ((b_i = 1) \implies \theta_i : \theta_i \in G_{\text{opt}})$
    $V' = V' \cup \{b_i\}$
    $G' = G' \cup \{\phi_i\}$
**end**
**let** $h = \text{maximize}(\sum_i b_i)$
**MAXSAT**$(G', V', h)$
**return** $\{\theta_i : b_i = 0\}$

## 5.5 Finding MIN-CONFLICT Sets from a MIN-REPAIR Set

In this section, we describe an algorithmic heuristics to compute conflict sets from minimal repairs.

Let $R$ be a minimal repair of $G$. Based on Corollary 1, we know that $R$ overlaps with some minimal conflicting sets. Our algorithm will identify subsets of $R$ as candidates of conflicting sets. For each identified candidate sets, we enlarge them incrementally until a conflicting set is found.

Given a set of goals, $X$, we will denote $V(X)$ as the set of variables used in $X$. First, we observe the following:

**Proposition 2.** *Suppose $C = C_1 \cup C_2$ and $V(C_1) \cap V(C_2) = \emptyset$. If $C$ is a conflicting set, then each $C_i$ is also conflicting.*

This means we can partition $R$ into $R = \{R_1, R_2, \dots\}$ based on the variables: $V(R_i) \cap V(R_j) = \emptyset$ for all $i \neq j$. Each $R_i$ is treated as a *candidate seed*, and will lead to the discovery of a conflicting set.

**Example 4.** *Consider the problem given in Example 1 and Example 2. The repair sets are:*

- $R_1 = \{g_2, g_3\}$
- $R_2 = \{g_1\}$

*The MAX-SAT will try to satisfy a maximal number of goals, and $R_2$ will be determined to be a minimal repair. We will use $R_2$ as the seed to generate a conflicting set. We then partition $R_2$ into subsets, each with non-overlapping variables. In this case, there is only one subset in the partition, $\{R_2\}$.*

Once we identify $R_i$ as the repair, we will augment it by one goal at a time until we find a conflict. We want to use is to add the most restrictive goal to be added at each time.

**Definition 7** (Incremental addition)**.** *Let $X \subset G$ be a subset of goals. We define* incremental addition *to $X$ as the goal $\phi_X$ in $\overline{X}$ such that $V(\phi_X)$ has the most overlap with $V(X)$.*

$$\phi_X = \underset{\theta \in G - X}{\operatorname{argmax}} |V(\theta) \cap V(X)| \tag{5.1}$$

We can now define a heuristic algorithm that constructions a conflicting set from the candidate seed $R_i$.

**Algorithm 2: grow-to-conflict**$(R_i)$

**Input:** A subset of goals $R_i$
**let** $C = R_i$
**while** $C$ *is satisfiable* **do**
    $C = C \cup \{\phi_C\}$
**end**
**return** $C$

Now, we can complete the algorithm that maps a minimal repair to one or more conflicting sets.

**Algorithm 3: find-conflicting-sets**$(R)$

**Input:** A minimal repair set $R$
**let** $\mathcal{C} = \emptyset$
**for** $R_i \in$ parition$(R)$ **do**
  **let** $C =$ **grow-to-conflict**$(R_i)$
  $\mathcal{C} = \mathcal{C} \cup \{C\}$
**end**
**return** $\mathcal{C}$

**Example 5.** *In our running example, the minimal repair is found by MAX-SAT which is partitioned into just one subset* $\{g_1\}$.

*If we apply* **grow-to-conflict**$\{g_1\}$, *the two goals added will be:*

- $\phi_{\{g_1\}} = g_0$.
- $\phi_{\{g_1, g_0\}} = g_2$.

*Thus, the final conflicting set identified is:* $\{g_0, g_1, g_2\}$.

## 5.6 Case Study

In this section, we shall present a short case study using the methods presented in this chapter's prior sections. For this case study let us revisit *Segment Coloring* described in Problem 5.

**Example 6.** *From Problem 5, we construct the following relational constraint model* $\Theta$*:*

- *A global variable scope* $V = \bigcup_i \{s_i^s \in \mathbb{Z}\} \cup \{s_i^e \in \mathbb{Z}\} \cup \{s_i^c \in [0, 4)\}$.
- *A global goal scope* $G = \{g_1, g_2, g_3, g_4\}$.

For Example 6, we will extend Problem 5 to include an additional goal $g_5$ which will enforce a total length $t^l$ of all segments:

$$g_5 \leftarrow \sum_i s_i^l = t^l \tag{5.2}$$

The model $\Theta$ will contain three segments $s_1, s_2, s_3$ which will have predefined lengths $s_1^l, s_2^l, s_3^l$. Equation 5.2 shows that $g_5$ enforces a relationship between $t^l$ and the segment lengths such that if this conditions is not met the model $\Theta$ will become infeasible.

**Example 7.** *Let us present an infeasible condition for* $\Theta$*:*

- $t^l = 50$
- $s_1^l = 15$
- $s_2^l = 25$
- $s_3^l = 15$

Using Example 7, let us use our support operations defined in the previous section. We define the overall program model as the following:

$$\langle G, V \rangle = \left\langle \{g_1, g_2, g_3, g_4, g_5\}, \bigcup_{i=1}^{3} \{s_i^s \in \mathbb{Z}\} \cup \{s_i^e \in \mathbb{Z}\} \cup \{s_i^c \in [0, 4)\} \right\rangle$$

The first step is to find the minimum repair set from $G$ using Algorithm 1:

$$\begin{aligned} \text{repair}(G, V) &= R^* \\ \text{repair}(G, V) &= \{g_i : b_i = 0\} \\ &= \{g_5\} \\ \therefore R^* &= \{g_5\} \end{aligned}$$

Now that we have our minimum repair set $R^*$, the second step is to partition $R^*$ into subsets $R$ such that we can select a candidate seed set $R_i$ using the intuition from Corollary 1 and Proposition 2:

$$\begin{aligned} R^* &= \{g_5\} \\ \textbf{apply } \text{partition} &: R^* \mapsto R \\ \text{partition}(\{g_5\}) &= \{\{g_5\}\} \\ \textbf{apply } \text{select}_c &: R \mapsto R_i \\ \text{select}(\{\{g_5\}\})_c &= \{g_5\} \\ \therefore R_i &= \{g_5\} \end{aligned}$$

Since $|R^*| = 1$, $R^*$ itself can be the candidate $R_i$ and does not iterate to other candidates[1].

The third step is to build conflict sets from $R_i$ using **grow-to-conflict**$(R_i)$ (Al-

[1] detailed in Algorithm 3 iteration.

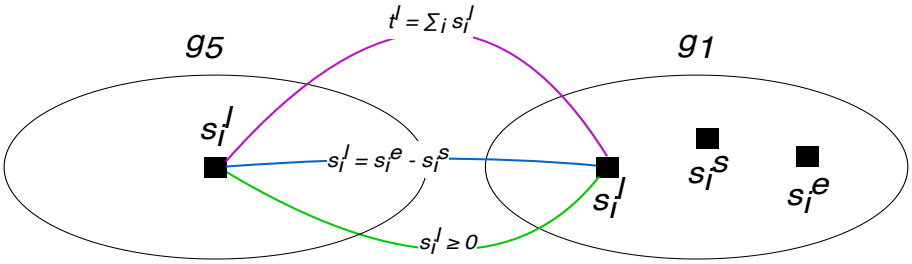



Figure 5.1: Conflict on $\{g_1, g_5\}$

gorithm 2) and the heuristic *By Overlap* (Equation 5.1):

$$\begin{aligned}
\phi_{\{g_5\}} &= \underset{\theta \in G - \{g_5\}}{\operatorname{argmax}} |V(\theta) \cap V(\{g_5\})| \\
&= g_1 \\
C &= \{g_1\} \cup \{g_5\} \\
&= \{g_1, g_5\}
\end{aligned}$$

Applying $\{g_5\}$ to Equation 5.1 finds a neighboring goal not in $\{g_5\}$ which has the most literals in common. In this case, $g_1$ is the only goal with related literals to $\{g_5\}$ so the resultant conflict set $C$ will be $\{g_1, g_5\}$. The literal relationships between $g_1$ and $g_5$ can be seen in Figure 5.1. Since there are no more partitions in $R$ we can assert that $\mathcal{C} = \{C\}$ as all conflicting sets.

In conclusion, the results of this case study as follows:

- MIN-REPAIR Set: $R^* = \{g_5\}$
- Conflicting Sets of $\Theta$: $\mathcal{C} = \{\{g_1, g_5\}\}$

Furthermore, this case study demonstrates the effectiveness of the application of our methods on CSPs, such as *Segment Coloring* (Problem 5).

# Chapter 6

# Experiments

We have presented our approach to our work in the previous chapter. We have shown methods of iterative and interactive debugging for data-driven CP models. In this chapter, we shall utilize a data built CSP to evaluate our methods. We also will evaluate various SAT solvers and demonstrate the reasoning behind our choice in the solver.

First, we will go over the CSP problem that we use in our evaluation along with our experimental setup, Section 6.1. Lastly, we will go over all the experiments and their results in detail, Section 6.2.

## 6.1 Implementation

In this section, we go over the overall setup for our experimental evaluation of our methods. First, by providing a formal definition of the **Random *k*-SAT** CSP. Then lastly, providing an overview of our technical setup for running our experiments.

### 6.1.1 The Random *k*-SAT Problem

Propositional CSPs consist of a CNF of DNF clauses with bitwise terms. The ***k*-SAT** Problem is one Propositional CSP which defines the size of each DNF clause. We can define a $k$-SAT CNF as the following:

$$C \leftarrow l_0 \wedge l_1 \wedge \ldots \wedge l_N \tag{6.1}$$

Where $C$ is the CNF and $L = \{l_0, l_1, \ldots, l_N\}$ are the disjunctive clauses. Each $l$ will have a $k$ number of literals that make up the disjunctive expression. We can see an example of an $l$ below:

$$l^k \leftarrow \bigvee_{v \in \nu}^{k} v \tag{6.2}$$

Where $v \in \nu$ are the member literals of the disjunctive expression $l$ and $\nu$ is the set of literals in the CSP model. Each $v$ have a domain of $\{0, 1\}$ and could consist of negations of these literals:

$$l^k \leftarrow \neg v_i \vee v_j \vee \neg v_z \vee \ldots \tag{6.3}$$

The **Random *k*-SAT** CSP is a variant of the $k$-SAT CSP, which have $L$ randomly generated [27]. The parameters includes $N$, $V$, $k$, and $p$:

| | |
|---|---|
| $N$ | Number of disjunctive clauses to be generated. |
| $V$ | Number of model variables to be generated. |
| $k$ | Size of each disjunctive clause to be generated. |
| $p$ | Probability of the clause literals not being negated. |

The members of each $l$ are chosen uniformly from $\nu$ [27] where a random factor of $p$ chooses the negations of these choices.

For our experimentation, we will use the Random $k$-SAT problem. This CSP

shall demonstrate the effectiveness of both the solver of choice and the MAX-SAT algorithm we use for our CP debugging methods.

### 6.1.2 Technical Setup

We ran our experiments on a system with 2 × Xeon(R) E5-2630 v3 @ 2.40GHz 8-core CPUs and 64 GB of RAM running **Ubuntu 20.04 LTS**. We implement our base experiment source in **Clojure 1.10**[1]. Our SAT solver of choice is **CP-SAT**[2] from Google's OR-Tools suite [42].

## 6.2 Evaluation

This section will take a look at the results found in the benchmarking done with our methods and supporting methods. We will start by looking at how the MAX-SAT problem performs with varying clauses $N$, literal sizes in clauses $k$, and variables sizes $V$. Next, we present our findings for our MIN-REPAIR operation under varying $N$, $k$ and $V$ including stats for all. Finally, we present findings for our MIN-CONFLICT operation comparing the algorithm's heuristics.

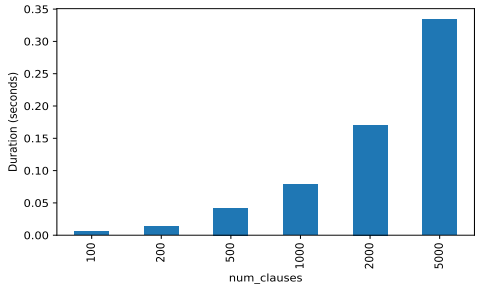
### 6.2.1 MAX-SAT

For MAX-SAT evaluation, we performed three rounds of experimentation. Each round, we varied $N$, $V$, and $k$ over a degree of bins. We measured the runtime duration (in seconds) for each bin for the MAX-SAT problem to be solved.

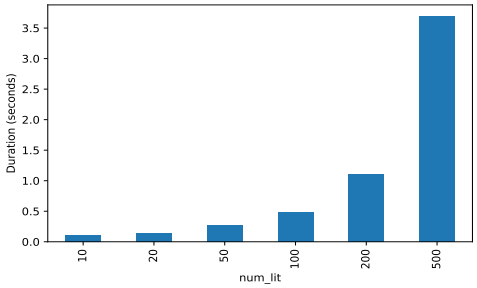
In our first round, we vary $N$ over $\{100, 200, 500, 1000, 2000, 5000\}$ which are the

[1]Clojure runs on the Java Virtual Machine (JVM). For more details on the Clojure programming language visit https://clojure.org/.

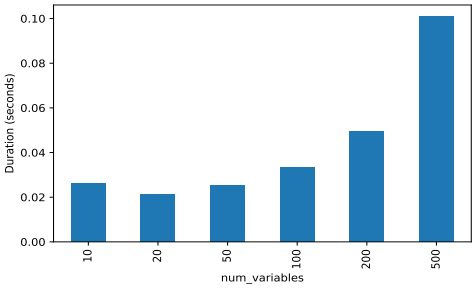
[2]CP-SAT is written in C++. For more details on this solver visit https://developers.google.com/optimization/cp.




(a) *Duration* over *Number of Clauses*.




(b) *Duration* over *Number of Clause Literals*.




(c) *Duration* over *Number of Variables*.

Figure 6.1: MAX-SAT Benchmarks

bins. $V$ and $k$ are constants of 10 and 3 respectively. We observe from this round that as $N$ grows over the bins, the duration only increases at a steady rate, as seen in Figure 6.1a.

In our second round, we vary $k$ over $\{10, 20, 50, 100, 200, 500\}$ which are the bins. $N$ is now constant with a value of 5000, the maximum value in the bins from the first round. Since $V$ must be greater than or equal to $k$ we decided to vary $V$ by $k \times 10$ which results to $\{100, 200, 500, 1000, 2000, 5000\}$. In this round, we found that the runtime duration of varying $k$ grew much larger when $k \geq 100$ than the durations found in the first round, as seen in Figure 6.1b.

In our third round, we vary $V$ over $\{10, 20, 50, 100, 200, 500\}$ which are the bins. We now have $N$ and $k$ as constants for this round with values 1000 and 5, respectively. We observe from the results of this round that the increasing variables have no effect

on the runtime duration until $V \geq 100$ where the duration proliferates even before reaching $V = 500$, as seen in Figure 6.1c.

To summarize these results, the MAX-SAT runtime duration scales gracefully as $N$ increases, however, scales much faster as $V$ and $k$ increases too. All results in Figure 6.1 show a polynomial growth in runtime. These findings are a joyous find for MAX-SAT as typically large CSPs will have more constraints than variables and constraint literals. With the MAX-SAT findings being satisfying, let us continue to MIN-REPAIR which will use MAX-SAT to find the least amount of unsatisfying constraints.

### 6.2.2 MIN-REPAIR

This section will go over the experimental results of the benchmarking done with our implementation of MIN-REPAIR. Similar to our benchmarking of the MAX-SAT implementation we ran MIN-REPAIR in three rounds for varying $N$, $V$, and $k$. We vary these parameters over a set of bins. For each bin, we measure the runtime duration (in seconds) which it took to run MIN-REPAIR.

In our first round, we vary $N$ over a set of bins $\{100, 200, 500, 1000, 2000, 5000\}$, same as the first round of MAX-SAT. $V$ and $k$ are constants with the values of 10 and 3, respectively. In this round, we observe the gradual polynomial growth that we have seen with MAX-SAT, see Table 6.1a and Figure 6.2a, with even 5000 randomly generated clauses finding a repair set within half of a second.

In our second round we vary $k$ over a set of bins $\{10, 20, 50, 100, 200, 500\}$ continuing the pattern from MAX-SAT benchmarking. As before, we also vary $V$ over $k \times 10$ due to the rule of $V \geq k$. $N$ is a constant with a value of 5000, the maximum from the first round. We observe in this round, seen in Table 6.1b and Figure 6.2b, that the runtime duration becomes even worse than in MAX-SAT and maximizes at 10

| | num_clauses | duration |
|---|---|---|
| **count** | 6.000000 | 6.000000 |
| **mean** | 1466.666667 | 0.131402 |
| **std** | 1865.118406 | 0.163400 |
| **min** | 100.000000 | 0.006937 |
| **25%** | 275.000000 | 0.023375 |
| **50%** | 750.000000 | 0.069048 |
| **75%** | 1750.000000 | 0.167592 |
| **max** | 5000.000000 | 0.434791 |

(a) Varying Clauses.

| | num_lit | duration |
|---|---|---|
| **count** | 6.000000 | 6.000000 |
| **mean** | 146.666667 | 2.793674 |
| **std** | 186.511841 | 3.632811 |
| **min** | 10.000000 | 0.305031 |
| **25%** | 27.500000 | 0.561936 |
| **50%** | 75.000000 | 1.328637 |
| **75%** | 175.000000 | 3.104134 |
| **max** | 500.000000 | 9.797402 |

(b) Varying Clause Literals.

| | num_variables | duration |
|---|---|---|
| **count** | 6.000000 | 6.000000 |
| **mean** | 146.666667 | 0.134009 |
| **std** | 186.511841 | 0.113054 |
| **min** | 10.000000 | 0.055149 |
| **25%** | 27.500000 | 0.062508 |
| **50%** | 75.000000 | 0.088157 |
| **75%** | 175.000000 | 0.148767 |
| **max** | 500.000000 | 0.349659 |

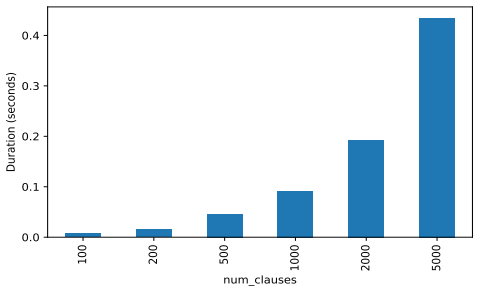
(c) Varying CP Variables.

Table 6.1: Repair Set Results.

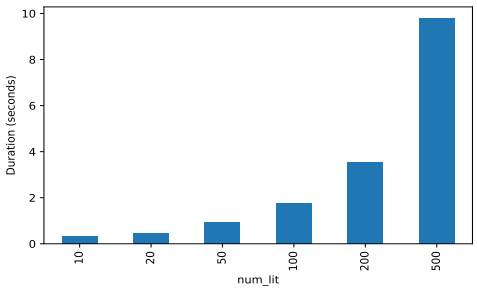
seconds with $k = 500$ further showing the complexity problem with large constraint clauses in the MAX-SAT model. This is ok however as 500 literals in a single clause is quite large and uncommon to see in CSPs.

In our third round we vary $V$ over a set of bins $\{10, 20, 50, 100, 200, 500\}$. $N$ and $k$ are constants in this round with values 1000 and 5 respectively. We observe in this round, seen in Table 6.1c and Figure 6.2c, that the runtime duration maintains the similar results to MAX-SAT where the duration remains steady until $V \geq 100$ then grows by larger quantities.

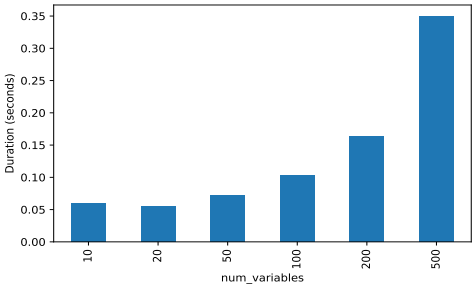
To summarize these results, the MIN-REPAIR runtime duration we see a similar pattern to the MAX-SAT results. These results are good news as we have a method for MIN-REPAIR which has gradual polynomial growth. Now let us process the




(a) *Duration* over *Number of Clauses*.




(b) *Duration* over *Number of Clause Literals*.




(c) *Duration* over *Number of Variables*.

Figure 6.2: MIN-REPAIR Benchmarks

results for MIN-CONFLICT which uses the MIN-REPAIR procedure.

### 6.2.3 MIN-CONFLICT

This section will go over the experimental results of the benchmarking done with our implementation of MIN-CONFLICT. We utilize two heuristics for the MIN-CONFLICT process:

- *Randomly*
- *By Overlap*

*By Overlap* is the hueristic we defined in Equation 5.1 we will denote as $\phi^o_X$. *Randomly* is a hueristic which builds the conflicting set randomly with Algorithm 2 and the *Randomly* hueristic equation $\phi^r_X$:

| | size_random | duration_random | size_overlap | duration_overlap |
|---|---|---|---|---|
| **count** | 50.000000 | 50.000000 | 50.000000 | 50.000000 |
| **mean** | 140.160000 | 1.146695 | 40.120000 | 0.304075 |
| **std** | 30.958405 | 0.286401 | 10.813144 | 0.089717 |
| **min** | 94.000000 | 0.715329 | 22.000000 | 0.168039 |
| **25%** | 117.000000 | 0.914972 | 32.250000 | 0.246340 |
| **50%** | 139.000000 | 1.157183 | 39.000000 | 0.282325 |
| **75%** | 153.750000 | 1.285555 | 44.750000 | 0.336757 |
| **max** | 238.000000 | 1.965448 | 66.000000 | 0.552254 |

Table 6.2: Conflict Set Results

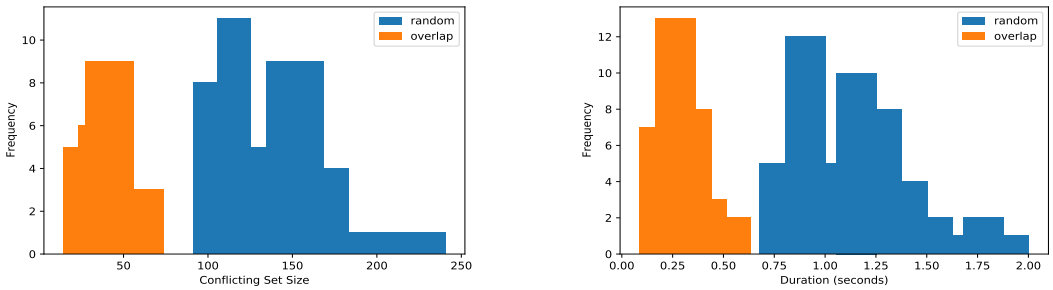
$$\phi_X^r = (G - X)_i : i \sim \mathrm{U}(0, |G - X|) \tag{6.4}$$

We use *Randomly* as a comparison heuristic to *By Overlap* in our benchmarking. We ran our implementation of Algorithm 3 on both heuristics over 50 iterations. In order to provide a more realistic case of conflicts to find, we create two $k$-SAT problems such that we have a satisfiable problem $P$ and an unsatisfiable $P'$:

- Parameters for $P$ include $N = 200$, $V = 1000$, $k = 3$, and $p = 0.5$.
- Parameters for $P'$ include $N = 100$, $V = 10$, $k = 3$, and $p = 0.5$.

We then create a merged problem $\rho$ where $\rho = P \cup P'$. This merge process creates a problem that will contain the clause conflicts in $P'$ but more clauses that are no conflicts.

From our findings benchmarking $\rho$ seen in Table 6.2, Figure 6.3a, and Figure 6.3b we can see that *By Overlap* runs much more efficiently than *Randomly* averaging 304.08 milliseconds (less than half a second) per iteration versus *Randomly* which is averaging at 1.15 seconds. Not only is duration lower for *By Overlap*, but the cardinalities of the conflict sets as well. Where *Randomly* averages in cardinality at about 140 constraints per conflict, *By Overlap* averages at only around 40 constraints per




(a) Conflicting Set Cardinalities.

(b) Conflicting Set Durations.

Figure 6.3: MIN-CONFLICT Benchmarks

conflict. These results are great because the fewer constraints found in the conflicts, the fewer fixes to be done to the clauses to create satisfiability.

# Chapter 7

# Related Work

This chapter will provide information on related works in the field of research behind this literature. We will show some influential and recent works done in SAT solvers and CP. We will also present works mostly focused on works done in Iterative and Interactive Debugging for CP.

## 7.1 SAT Solvers

In this section, we go over some notable SAT solvers which have some direct or indirect contribution towards the SAT engine we use in our work. Other known solvers include but are not limited to *MINISAT* [22], *Tinisat* [31], *FznTini* [32], and *BEE* [40]. More information on SAT solvers can be found in recent surveys [2, 43].

**OR-Tools CP-SAT [42].** CP-SAT is an open-sourced solver that is part of the OR-Tools [1] suite of Operations Research (OR) technologies developed by Perron L. and Furnon V. at Google. CP-SAT is one of the most modern efficient SAT solvers to date

[1]The website and resources for the OR-Tools suite can be found at https://developers.google.com/optimization/.

and uses various modern SAT solver components to achieve this [3,6,17,25,38,52]. It has won the top three places in its ability to solve many of the problems in the recent MiniZinc challenges [49,50]. The efficiency and modernization in the implementation of CP-SAT are why we choose to target our work.

**Glucose [6].** Glucose is a well known open-sourced SAT solver heavily based on another solver called *MINISAT* [22]. Glucose was created by Audemard G. and Simon L. in collaboration between University Lille-Nord de France and University Paris-Sud. A revolutionary Boolean SAT Solver for the modern age which improves the Conflict Directed Clause Learning (CDCL) algorithms by identifying certain clauses called 'Glue Clauses' when performing propagation.

**Chaff [41].** One of the more well known SAT solvers, Chaff was developed by Moskewicz M., Madigan C., Zhao Y., Zhang L., and Malik S. as a collaboration between University of California (UC Berkeley), Massachusetts Institute of Technology (MIT), and Princeton University. Chaff is a revolutionary Boolean SAT Solver for its time in its efficiency. It achieved this by the optimization of Boolean Constraint Propagation (BCP) algorithm and the Variable State Independent Decaying Sum (VSIDS) decision heuristics.

## 7.2 Iterative and Interactive Debugging

In this section, we go over some previous Iterative and Interactive Debugging works done. These works consist of prior techniques for debugging constraints, providing visualizations for representing constraints and their relationships, and annotating constraints as data.

**A Visualization Tool for Constraint Program Debugging. [28]** This work is done by authors F. Goualard and F. Benhamou from Institut de Recherche en Informatique de Nantes in Nantes, France. In this work, the authors propose a conceptual solution for debugging Constraint Programming by allowing the CNF to be reduced to a smaller set, then creating visualization(s) off of these constraints. The authors refer this constraint set as an S-Box.

**CLPGUI: A Generic Graphical User Interface for Constraint Logic Programming. [23]** Constraint Logic Programming Graphical User Interface (CLPGUI) is a constraint programming visualization suite created by F. Fages, S. Soliman, and R. Coolen from Projet Contraintes in Le Chesnay, France. CLPGUI provides both user interfaces and visualizations for Constraint Programming systems. CLPGUI is demonstrated by using the Constraint Programming systems GNU-Prolog [18] and SICStus-Prolog [8], however, the authors claim it can be ported to be used with other Constraint Programming systems as well. The system generates the constraint system model's visualizations by sending model data over to CLPGUI using XML data structures when solving the solution space. CLPGUI displays annotations for the variables specified by the Constraint Programming system, which allows the user to distinguish the variables being visualized by CLPGUI.

## 7.3 Using SAT in the Relational Model

In this section, we will go over a few works that aim to solve the objective of the work done in Chapter 4.

**Combining relational algebra, SQL, constraint modelling, and local search. [7]** Marco Cadoli and Toni Mancini wrote this literature at Università di Roma in Rome,

Italy. In this work, the authors argue the importance of combining relational algebra and SQL with constraint modelling and local search for large NP-complete problems. As a result of this argument, they propose `CONSQL` which takes on a similar role to SAT/SQL introduced in Chapter 4 of this literature.

**Solving SQL Constraints by Incremental Translation to SAT. [37]** This literature was written by Robin Lohfert, James Lu, and Dongfang Zhao at Emory University in Atlanta, GA, USA. For this work, the authors demonstrate that by using a technique, one can implement an engine for the query language, `CONSQL` [7].

# Chapter 8

# Conclusions

In this chapter, we will summarize the contributions the work in this literature adds to relational databases and iterative and interactive debugging in constraints.

## 8.1 Summary

In summary we provide the following contributions:

- A Relational Constraint Framework for supporting data-driven constraint programming in the relational model using the SQL extension SAT/SQL.
- A set of Iterative and Interactive Relational Constraint Debugging methods which provide the following:
  - A process using MAX-SAT modelling for finding a minimum repair set which is a need for performing automatic constraint repair.
  - Building conflict sets efficiently by using the found minimum repair set.

Using SAT/SQL, we see the potential for creating a simplified modelling language for performing CP in relational data sources and the data pipeline. We see from

Chapter 6, positive results in using our CP debugging operators in the relational constraint model. From these same results, we see that MIN-REPAIR using MAX-SAT can perform efficiently over lots of constraints in polynomial time to find a minimum repair set. Expanding on this, we also provided a procedure for MIN-CONFLICT operator which builds conflict sets more efficiently than naive approaches [51] and approaches used in related work [28].

## 8.2 Future Work

This section will go over the future work we hope to see a rise from work done in our literature. Possible continuing work includes:

- We would like to see static analysis and query optimization to be done on the language extension SAT/SQL to ensure proper standardization and implementation of the SAT/SQL.
- We would like to see an assessment of SAT/SQL in the effectiveness of syntax using the number of lines, speed of writing, and an average number of errors introduced into the constraint program.
- We would like to see an implementation of visualizations for the support operators' results to cover the interactive component of the debugging process. These visualization implementations could be similar to the ones in use from previous works [23, 28].

# Appendices

# Appendix A

# Relational Data Model Extras

## A.1 CREATE Databases and Schemas

We can use a `CREATE` SQL statement to create a database named *HumanResources*:

```
CREATE DATABASE HumanResources;
```

Schemas have the same statement as databases only with the keyword `SCHEMA` instead of `DATABASE`:

```
CREATE SCHEMA HiringManagers;
```

## A.2 Primary and Foreign Keys

The use of keys in relational DBMSs is not quite the same as keys in the Hash Table data structure, which is more similar to indices in relational DBMSs. The main keys known in relational DBMSs are **Primary Keys** and **Foreign Keys**. Let us begin with Primary Keys.

A *Primary Key* is an attribute or set of attributes which uniquely identifies tuples within a relation. This key creates a constraint known as an *entity integrity* constraint

that enforces that all value sets within the Primary Key must be unique and will not allow tuples to be inserted into the relation if it would violate this constraint. There are two other keys which derive from this one. A *surrogate key* is a key which is only useful at uniquely identifying tuples rather than having inherent meaning behind it. A *composite key* can uniquely identify tuples by using two or more attribute combinations. The attribute *id* from Table 3.2 makes a good candidate for a Primary Key as all its values uniquely identify each of the tuples. To do this in SQL, we can create a primary key which includes the attribute *id*:

```
CREATE TABLE Persons(
    id INT NOT NULL PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    company TEXT
);
```

We can also define the primary key as a constraint:

```
CREATE TABLE Persons(
    id INT NOT NULL,
    first_name TEXT,
    last_name TEXT,
    company TEXT,
    PRIMARY KEY (id)
);
```

The primary key constraint can also be named, such as '*PK_Person*':

```
CREATE TABLE Persons(
    id INT NOT NULL,
```

```
    first_name TEXT,
    last_name TEXT,
    company TEXT,
    CONSTRAINT PK_Person PRIMARY KEY (id)
);
```

A *Foreign Key* (or *Reference Key*) is an attribute which is shared with another relation creating a relationship between the relations that share this attribute. This key creates a constraint known as an *referential integrity* constraint that enforces that all values within this attribute must in the tuples of the relation which the Foreign Key is pointing to. To illustrate this let us say that $a$ is an attribute in relation $T$ that is a Foreign Key and that $a'$ is the target attribute in relation $T'$. This means that $t_1[a], t_2[a], \ldots, t_n[a]$ must contain values in $t'_1[a'], t'_2[a'], \ldots, t'_{n'}[a']$ if tuples $t \in T$ and tuples $t' \in T'$. Though Foreign Keys does not imply entity integrity constraints as Primary Keys do, they can still also be Primary Keys which enforce entity integrity constraints. This means that as long as a Foreign Key is not a Primary Key you can have duplicate values.

## A.3 Normalization

The normalization of database relations is the process of following strict group rules, known as normal forms, set for the structure and integrity of the data within. It was proposed by Codd as to allow for data to be queried and manipulated by using a language defined in first-order logic [11]. The common normal forms are Unnormalized Form (UNF), First Normal Form (1NF) [11], Second Normal Form (2NF) [12], Third Normal Form (3NF) [12], and Boyce-Codd Normal Form (BCNF) [13]. For our work we will focus on relations with normal forms up to 3NF and BCNF. For more details

on these normal forms and more normal forms see Chapter 19 of Ramakrishnan et. al. [44] and Chapter 8 Connolly et. al. [16].

## A.4 Other Relational Data Formats

As mentioned in Chapter 3, there are other forms of data storage which follow the Relational Data Model that are not relational DBMSs. Another form of data which follows the relational data model is CSV files. CSV contain values separated by commas in rows separated by newlines, hence the name 'Comma Separated Values.' The lines in CSV files would be the tuples, and the columns of values separated by commas would be the attributes. In CSV files, unlike relational databases, the files themselves are considered the relations. CSV files will sometimes have a header row where each of the values is the attribute names.

There are more methods of using the relational data model. For example, one can use Object data sources in a relational format. In this case, one would typically set up a collection of objects where each object is considered a tuple. The keys or member accessors of the object would be the attributes. The overall data structure would be one relation. This could be implemented in a DBMS such as MongoDB [1] or a data files such as JavaScript Object Notation (JSON) [2] files.

The advantage of using these other data formats is that scalability is less complicated, meaning one does not have to construct structure before constructing the database. However, this comes as a double-edged sword, where the drawback is that it does not enforce well-structured data, which can lead to dirty data and lack of integrity within the data. Data sources such as MongoDB and JSON files do not enforce the relational data model either, which means they can be using an entirely

[1] https://www.mongodb.com/
[2] https://www.json.org/

different data model. Therefore with these methods, it is up to the client side's programmer to these data sources if a data model such as the relational data model is to be enforced.

# Appendix B

# Aggregation Queries

Let us go into the additional clauses we can use with our `SELECT` statements to aggregate our results. We will use Table B.1 for our target relation called *Employees*.

## B.1 DISTINCT Clause

The `DISTINCT` clause is an additional operation to the `SELECT` statement to grab only tuples that would have unique values in the selected attributes. This clause can be expressed in relational algebra. However, there is no need to ensure uniqueness as relational algebra implies this already. For example, let us say we wanted to grab all the values of *country*. In relational algebra, we only need to do the following:

$$\pi_{country}(Employees)$$

In a SQL `SELECT` statement, however, this would give duplicate tuples. With the `DISTINCT` clause, the result set would be the same as the above relational algebra expression. We see here the SQL equivalent of this relational algebra expression, with Table B.2 being the result set of this query:

| *id* | *name* | *company* | *country* |
|---|---|---|---|
| 5436 | Imogene J. Stephenson | Donec Egestas PC | United States |
| 2431 | Aurelia Y. Anderson | Auctor Nunc Corporation | United States |
| 8769 | Clementine U. Mcintyre | Ac Libero Nec Institute | Chile |
| 7445 | Renee E. Barry | Dictum Eu Inc. | Korea, South |
| 7359 | Belle B. Britt | Purus Maecenas Associates | Canada |

Table B.1: Relation of Employees called *Employees*.

```
SELECT DISTINCT country
FROM Employees;
```

## B.2 LIMIT Clause

A `LIMIT` clause is appended to the SQL `SELECT` statement when wanting to get a certain number of $n$ tuples. An example of the `LIMIT` clause is shown below with a result set shown in Table B.3:

```
SELECT DISTINCT country
FROM Employees
LIMIT 3;
```

## B.3 Aggregate Functions

In SQL `SELECT` statements we can perform mathematical aggregation operations on individual attributes or entire tuples. These operations in SQL include the following:

- `MAX` - Results in the maximum value of all tuples possible for given attribute.
- `MIN` - Results in the minimum value of all tuples possible for given attribute.
- `COUNT` - Results in the count of all tuples possible, optionally for given attribute.

| *country* |
|---|
| United States |
| Chile |
| Korea, South |
| Canada |

Table B.2: Result set of distinct results from *Employees*.

| *country* |
|---|
| United States |
| Chile |
| Korea, South |

Table B.3: Result set limited to 3 records from *Employees*.

- `SUM` - Results in the summation of all tuples possible for given attribute.
- `AVG` - Results in the average value of all tuples possible for given attribute.

Notice the phrase *all tuples possible*, this statement implies that the number of tuples in the same aggregation depends on how one is to set up the rest of SQL statement. For example, the `COUNT` aggregation applied to all attributes which will result in the total number of tuples in the source relation:

```
SELECT COUNT(*) AS num_employees
FROM Employees;
```

However, we see a statement with the same `COUNT` aggregation applied, but with a `GROUP BY` aggregation clause applied *country* attribute:

```
SELECT country, COUNT(*) AS num_employees
FROM Employees
GROUP BY country;
```

This statement will therefore result in several tuples for every *country* value.

## B.4 GROUP BY Clause

The `GROUP BY` clause in SQL statements will group tuples based on an attribute. This clause is mainly used in sceneries, where an aggregate operation is being performed.

## B.5 JOIN Clauses

`JOIN` clauses are when one wants to combine multiple relations in a single result set by a common attribute. There are several joins which can be performed between multiple, such as:

- `INNER JOIN` - Result set contains the intersecting tuples of the multiple relations.
- `LEFT JOIN` - Result set contains all tuples from first relation with intersecting tuples from the second.
- `RIGHT JOIN` - Inverse of the `LEFT JOIN`.
- `FULL JOIN` - Result set contains all tuples from all relations.

Consider an additional conceptual relation named *Companies* with attributes *id* and *company_name*. Now imagine that attribute *company* in relation *Employees* was actually a numeric company ID. We want to display all the tuples from our *Employees* relation but only need the tuples in *Companies*, which allows us to fetch the company names for the *Employees* tuples. For this we can use a `LEFT JOIN`:

```
SELECT Employees.name, Companies.company_name
FROM Employees
LEFT JOIN Companies
ON Employees.company=Companies.id;
```

A unique form of join called a `NATURAL JOIN` connects multiple relations by an already known connection between attributes with matching characteristics (same name and data type) or an equality already specified in the `WHERE` clause.

## B.6 UNION Clause

The `UNION` clause provides the ability to fetch a result set with all tuples in from two other result sets. This statement is written by using two `SELECT` statements. There are some restrictions to using this clause. The restrictions are as follows:

- The two source result sets must have the same number of attributes selected.
- The data types of these attributes must be similar in nature.
- The ordering of the attributes must match one another in each result set.

Consider an additional conceptual relation named *Clients* which has similar attributes to *Employees*. We can use `UNION` to merge the result sets:

```
SELECT name, company, country
FROM Employees
UNION
SELECT name, company, country
FROM Clients;
```

# Bibliography


[1] The "Send More Money" Problem. In *Constraint-Based Agents: An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*, A. Nareyek, Ed., Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2001, pp. 139–139.

[2] Alouneh, S., Abed, S., Al Shayeji, M. H., and Mesleh, R. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. *Artificial Intelligence Review 52*, 4 (Dec. 2019), 2575–2601.

[3] Ansótegui, C., Bonet, M. L., and Levy, J. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Theory and Applications of Satisfiability Testing - SAT 2009* (Berlin, Heidelberg, 2009), O. Kullmann, Ed., Lecture Notes in Computer Science, Springer, pp. 427–440.

[4] Arora, S., and Barak, B. *Computational Complexity: A Modern Approach*, 1st ed. Cambridge University Press, New York, NY, USA, 2009.

[5] Assi, M., and Haraty, R. A. A Survey of the Knapsack Problem. In *2018 International Arab Conference on Information Technology (ACIT)* (Nov. 2018), pp. 1–6.

[6] Audemard, G., and Simon, L. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Twenty-First International Joint Conference on Artificial Intelligence* (June 2009).

[7] Cadoli, M., and Mancini, T. Combining relational algebra, SQL, constraint modelling, and local search. *Theory and Practice of Logic Programming 7*, 1-2 (Jan. 2007), 37–65.

[8] Carlsson, M., and Mildner, P. SICStus Prolog – the first 25 years. *arXiv:1011.5640 [cs]* (Nov. 2010).

[9] Chamberlin, D. D. Early History of SQL. *IEEE Annals of the History of Computing 34*, 4 (Oct. 2012), 78–82. Conference Name: IEEE Annals of the History of Computing.

[10] Clarke, E., Kroening, D., Sharygina, N., and Yorav, K. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2005), N. Halbwachs and L. D. Zuck, Eds., Lecture Notes in Computer Science, Springer, pp. 570–574.

[11] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (June 1970), 377–387.

[12] Codd, E. F. Further Normalization of the Data Base Relational Model. *Research Report / RJ / IBM / San Jose, California RJ909* (1971).

[13] Codd, E. F. Recent Investigations in Relational Data Base Systems. In *ACM Pacific* (1974).

[14] Codd, E. F. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[15] Codognet, P., and Diaz, D. Compiling constraints in clp(FD). *The Journal of Logic Programming 27*, 3 (June 1996), 185–226.

[16] Connolly, T., Begg, C. E., and Holowczak, R. *Business database systems*. Addison-Wesley, Harlow, England ; New York, 2008.

[17] Davies, J., and Bacchus, F. Solving MAXSAT by Solving a Sequence of Simpler SAT Instances. In *Principles and Practice of Constraint Programming – CP 2011* (Berlin, Heidelberg, 2011), J. Lee, Ed., Lecture Notes in Computer Science, Springer, pp. 225–239.

[18] Diaz, D., and Codognet, P. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming 2001*, 6 (2001).

[19] D'Silva, V., Kroening, D., and Weissenbacher, G. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27*, 7 (July 2008), 1165–1178. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[20] E, F. Codd, E. F. Derivability, redundancy and consistency of relations stored in large data banks. *ACM SIGMOD Record 38*, 1 (Aug. 1969), 17–36.

[21] Epstein, D. On the NP-completeness of cryptarithms. *ACM SIGACT News 18*, 3 (Apr. 1987), 38–40.

[22] Eén, N., and Sörensson, N. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing* (Berlin, Heidelberg, 2004), E. Giunchiglia and A. Tacchella, Eds., Lecture Notes in Computer Science, Springer, pp. 502–518.

[23] Fages, F., Soliman, S., and Coolen, R. CLPGUI: A Generic Graphical User Interface for Constraint Logic Programming. *Constraints 9*, 4 (Oct. 2004), 241–262.

[24] Fontaine, D. *The Art of PostgreSQL*, 2nd ed., vol. 1. Dimitri Fontaine, 2019.

[25] Fu, Z., and Malik, S. On Solving the Partial MAX-SAT Problem. In *Theory and Applications of Satisfiability Testing - SAT 2006* (Berlin, Heidelberg, 2006), A. Biere and C. P. Gomes, Eds., Lecture Notes in Computer Science, Springer, pp. 252–265.

[26] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

[27] Gent, I. P., and Walsh, T. The SAT Phase Transition. In *In Proc. ECAI-94* (1994), pp. 105–109.

[28] Goualard, F., and Benhamou, F. A visualization tool for constraint program debugging. In *14th IEEE International Conference on Automated Software Engineering* (Oct. 1999), pp. 110–117.

[29] Greenwald, R., Stackowiak, R., and Stern, J. *Oracle Essentials: Oracle Database 12c*. "O'Reilly Media, Inc.", Sept. 2013.

[30] Gödel, K. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik 37*, 1 (Dec. 1930), 349–360.

[31] Huang, J. A Case for Simple SAT Solvers. In *Principles and Practice of Constraint Programming – CP 2007* (Berlin, Heidelberg, 2007), C. Bessière, Ed., Lecture Notes in Computer Science, Springer, pp. 839–846.

[32] Huang, J. Universal Booleanization of Constraint Models. In *Principles and Practice of Constraint Programming* (Berlin, Heidelberg, 2008), P. J. Stuckey, Ed., Lecture Notes in Computer Science, Springer, pp. 144–158.

[33] Ivančić, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., and Ashar, P. F-Soft: Software Verification Platform. In *Computer Aided Verification* (Berlin, Heidelberg, 2005), K. Etessami and S. K. Rajamani, Eds., Lecture Notes in Computer Science, Springer, pp. 301–306.

[34] Kanellakis, P. C., and Goldin, D. Q. Constraint programming and database query languages. In *Theoretical Aspects of Computer Software* (Berlin, Heidelberg, 1994), M. Hagiya and J. C. Mitchell, Eds., Lecture Notes in Computer Science, Springer, pp. 96–120.

[35] Krajicek, J. *Bounded Arithmetic, Propositional Logic and Complexity Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1995.

[36] Kumar, V. Algorithms for constraint-satisfaction problems: A survey. *AI magazine 13*, 1 (1992), 32–32.

[37] Lohfert, R., Lu, J. J., and Zhao, D. Solving SQL Constraints by Incremental Translation to SAT. In *New Frontiers in Applied Artificial Intelligence* (Berlin, Heidelberg, 2008), N. T. Nguyen, L. Borzemski, A. Grzech, and M. Ali, Eds., Lecture Notes in Computer Science, Springer, pp. 669–676.

[38] Lopez-Ortiz, A., Quimper, C.-G., Tromp, J., and Van Beek, P. A fast and simple algorithm for bounds consistency of the all different constraint. In *Proceedings of the 18th international joint conference on Artificial intelligence*

(Acapulco, Mexico, Aug. 2003), IJCAI'03, Morgan Kaufmann Publishers Inc., pp. 245–250.

[39] Mayoh, B., Tyugu, E., and Uustalu, T. Constraint Satisfaction and Constraint Programming: A Brief Lead-In. In *Constraint Programming* (Berlin, Heidelberg, 1994), B. Mayoh, E. Tyugu, and J. Penjam, Eds., NATO ASI Series, Springer, pp. 1–16.

[40] Metodi, A., Codish, M., and Stuckey, P. J. Boolean Equi-propagation for Concise and Efficient SAT Encodings of Combinatorial Problems. *Journal of Artificial Intelligence Research 46* (Mar. 2013), 303–341.

[41] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference* (Las Vegas, Nevada, USA, June 2001), DAC '01, Association for Computing Machinery, pp. 530–535.

[42] Perron, L., and Furnon, V. OR-Tools, July 2019.

[43] Prasad, M. R., Biere, A., and Gupta, A. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer 7*, 2 (Apr. 2005), 156–173.

[44] Ramakrishnan, R., and Gehrke, J. *Database management systems*, 3rd ed ed. McGraw-Hill, Boston, 2003.

[45] Rivin, I., Vardi, I., and Zimmermann, P. The n-Queens Problem. *The American Mathematical Monthly 101*, 7 (Aug. 1994), 629–639.

[46] Rosen, K. H. *Discrete mathematics and its applications*, 7th ed ed. McGraw-Hill, New York, 2012.

[47] Rossi, F., van Beek, P., and Walsh, T. *Handbook of Constraint Programming.* Elsevier Science Inc., USA, 2006.

[48] Staff, N. R. C. *Funding a Revolution: Government Support for Computing Research.* National Academies Press, Washington, 1999.

[49] Stuckey, P. J., Becket, R., and Fischer, J. Philosophy of the MiniZinc challenge. *Constraints 15*, 3 (July 2010), 307–316.

[50] Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., and Fischer, J. The MiniZinc Challenge 2008–2013. *AI Magazine 35*, 2 (June 2014), 55–60. Number: 2.

[51] Valdron, M., and Pu, K. Q. Data Driven Relational Constraint Programming. In *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)* (Aug. 2020), pp. 156–163.

[52] Zhang, L., Madigan, C., Moskewicz, M., and Malik, S. Efficient conflict driven learning in a Boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)* (Nov. 2001), pp. 279–285.