On the Utility of Adding An Abstract Domain and Attribute Paths to SQL

by

Weicong Ma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Albeit its popularity today, Relational Database Management Systems (RDBMSs) and the relational model still have many limitations. For example, one needs to pay premature attention to naming issues in the schema designing phase; and the syntax for conjunctive queries is verbose and redundant, especially for multi-table joins and composite primary/-foreign keys. In this thesis, we introduce and explain the method to handle and resolve these issues that is proposed by Borgida, Toman, and Weddell [3]: the conceptual schema that supports abstract relations and attributes, and an extended query language SQL^{path} built on top of standard Structured Query Language (SQL) that supports the usage of attribute paths and abstract attributes in queries. We demonstrate a systematic approach to map a database schema expressed in the relational model to the abstract relational model and illustrate how to write SQL^{path} queries with attribute paths to solve query problems involving complex table joins. This thesis can serve as both an introduction and tutorial to abstract database modelling and the SQL^{path} query language.

Additionally, we performed an empirical experiment to evaluate the performance of SQL^{path} when solving real database query problems by employing students with prior experience with SQL to read and write SQL^{path} queries and recorded their accuracy and time consumption against usage of regular SQL. The result of this experiment is presented in this thesis, including a statistical analysis of the results. In short, we uncover evidence that SQL^{path} is more efficient to use for both reading and writing conjunctive and alike queries, especially for non-trivial cases where multiple constraints were required. However, while SQL^{path} can hide explicit table joins when writing queries spanning multiple intermediate tables, whether this benefit can make users produce more accurate results still remains unclear as we were not able to draw any conclusion from collected data due to lack of statistical significance.

Acknowledgements

First of all, I would like to show my sincere gratitude and appreciation to my supervisor Prof. Grant Weddell. He suggested this idea at the very beginning and guided me throughout my master study. He is always patient and conscientious. I really learn a lot and improve a lot. Without his support, I could not make this thesis possible.

Second, I would like to thank my co-supervisors Prof. Wayne Oldford and Prof. David Toman. Prof. Oldford helped me on designing the experiment and statistical analysis while Prof. Toman provided great insight and advices throughout the progress of my thesis.

Last but not the least, I would also like to thank the readers of my thesis, Prof. Semih Salihoglu and Prof. Mei Nagappan, for their constructive and valuable feedback.

Dedication

This is dedicated to my parents and sister for their unconditional love and support. Thanks to my boyfriend Shengying Pan who is always by me side helping me and encouraging me. Moreover, thanks to my dog Venus for letting me be a part of her life. Finally, I would like to take this opportunity to thank all my friends for all the company and sharing all the tears and joys.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANSI American National Standards Institute 9

DDL Data Definition Language 9

DML Data Manipulation Language 9

ISO Organization for Standardization 9

ORM Object Relational Mapping 1

RDBMS Relational Database Management System iii, 1, 2, 4, 11, 18, 20, 31, 67

SQL Structured Query Language iii, 1, 3, 4, 8–14, 29–33, 66, 67

Chapter 1

Introduction

Many new database management systems and different technologies are emerging lately, but an RDBMS is still the mostly used when dealing with real world data storage problems. With an RDBMS, entity relationship notation is applied to design a relational model and derive relational database schemata. Although this provides us a tool with object-centered modelling notation to construct semantic data models [12], an RDBMS still faces many limitations and problems, albeit its popularity. For example, the need to prematurely commit to an "external key" (printable values) in designing relational schemas; and the need to choose a single and simple way to refer to all entities/tuples in a class/table rather than allow for variations [17]. Oftentimes, one needs to coin an attribute that is not part of an object as its key to build relationships, and this will in turn require extra attention from SQL developers to not forget them when writing queries, especially for conjunctive and alike queries where table joins are extensively applied.

To help with these problems, many new models and tools have been proposed to extend the usage of relational model and standard SQL. For instance, software engineers have used Object Relational Mapping (ORM) tools to create a virtual object database on top of an RDBMS to use within programming language to hide the detail of primary key and foreign key constraints and perform database queries by invoking function methods on objects rather than joining the keys explicitly. However, this does not solve the problem, but only shifts the complexity to ORM developers. Moreover, as a programming language and database system specific solution, none of the ORM tools can be universally applied due to the differences between RDBMS engines.

On the other hand, recent work by Borgida, Toman and Weddell has proposed a new abstract relational model and an extension to SQL, called SQL^{path}, that allows implicit

foreign key joins in the form of "path expressions" [3]. Built on top of their earlier work in description logics [11] [18] [16] [19] [17], they have claimed that the addition of the "path expression" would make SQL programming more comfortable. As conjunctive and alike queries are heavily used for data retrieval from RDBMS, the use of SQL^{path} and the abstract relational model is particularly beneficial as it significantly simplifies the syntax of table joins by avoiding explicit column references. Moreover, as a lower level solution, this extension can be integrated into the database engine directly, while providing users a uniform interface to take advantage of.

However, they only provided definitions of the abstract relational model and SQL^{path}. There lacks a systematic way to convert database schema from the relational model to the abstract relational model and detailed explanation of how SQL^{path} can be used to write database queries. As a result, people who are interested in their work may not know how to apply it to real world applications. Additionally, although they expected SQL^{path} to be more efficient and accurate to use, this expectation is purely derived from assumptions and theoretical analysis. Whether its benefits and advantages can lead to actual performance improvement in real world applications was open.

To better understand and answer these questions, we first explain the abstract relational model and SQL^{path} in a more detailed fashion in this thesis. We present a systematic way to map relational model to abstract relational model, and demonstrate how to write database queries with SQL^{path}'s attribute path with detailed steps and examples, with the main focus on conjunctive queries. Additionally, we conducted an empirical experiment to test and evaluate the performance of the abstract relational model and SQL^{path} in real world settings: used by people writing real database queries to perform actual tasks. It's interesting to discover whether SQL^{path} can make conjunctive and alike queries easier to understand and design, especially in complex settings where multiple tables need to be joined to answer questions that involve a lot of constraints specified on multiple levels.

1.1 Contributions

We extend Borgida, Toman and Weddell's work on the abstract relational model and SQL^{path} in this thesis. We provide a detailed explanation of their core concepts and a systematic approach to map between the relational model and the abstract relational model, and how to write conjunctive queries with SQL^{path}. As a result, this thesis can be used as an introductory guide and tutorial to learn how to apply the abstract relational model and SQL^{path} in real world applications for people who are interested in them.

Moreover, we conducted the first empirical study to evaluate the abstract relational model and SQL$^{\text{path}}$ on comprehending and writing database queries with the main focus on conjunctive queries. We recruited real people to work on database query problems with both SQL and SQL$^{\text{path}}$ and compared their performance in two metrics: time consumption and correctness, to have a better understanding of the actual benefits of SQL$^{\text{path}}$'s added features. We applied statistical analysis to our collected data and concluded that the abstract relational model and SQL$^{\text{path}}$ is more efficient to use, for both query reading and writing questions, while the correctness difference is not statistically significant enough. These analytical results not only can help researchers to find applications that are more suitable for the abstract relational model and SQL$^{\text{path}}$, but also give them future research direction to further extend these tools to cover their weakness and enhance their strength.

1.2 Thesis Outline

The rest of this thesis is organized as follows:

In Chapter 2, we go through background knowledge on the relational database model and SQL. We illustrate how people use them to create entities and relationships to model real world applications, and how primary keys and foreign keys are used to identify objects and enforce constraints on relationships. Additionally, we explain how SQL is used to perform conjunctive and alike queries and discuss the problems of this traditional approach.

In Chapter 3, we introduce the abstract relational model. We first explain its two core concept of abstract domain and referring expression types, then demonstrate how to map a database schema from the relational model to the abstract relational model, with the help of referring expression type assignment. In the end, we explain how SQL$^{\text{path}}$ and its added attribute path work, and show how to use them to simplify the writing of database queries.

In Chapter 4, we go through the design and detailed experiment procedure of our empirical study at University of Waterloo to evaluate and compare the performance of the abstract relational model and SQL$^{\text{path}}$ against the relational model and SQL. We use statistical analysis to interpret our data collected from the experiment and present our hypotheses, evaluation and statistical conclusion in this chapter.

In Chapter 5, we conclude our work and recommend future research directions.

Chapter 2

Preliminaries

Before diving any further into the abstract relational model and SQL$^{\text{path}}$, we would like to first review the relational database model and SQL. In this chapter, we explain the core concepts of the relational model, such as how to declare relational tables to model real world applications, and the usage of primary key constraints and foreign key constraints to identify objects and establish relationships between entities. Moreover, we demonstrate how SQL queries can be used to retrieve information from database with examples.

2.1 The Relational Model

The relational data model was first proposed by Ted Codd from IBM in 1970 [7]. It uses the mathematical relation as its basic building block, and first order predicate logic as its base [8]. The relational data model is the primary data model for the majority of data storage applications today due to its simplicity and its mathematical foundation [14]. All of today's popular commercial and open source RDBMSs implement the relational model, such as DB2, Oracle, SQLServer, MySQL, and PostgreSQL.

A relational database consists of a collection of tables. A table stores rows of data values, where a row commonly connects to a real world data entity or relationship. The column names are used to help users interpret the stored data of entities and relationships, where each column corresponds to an attribute, and all values stored in a column are of the same data type, for example, `INT`, `STRING`, etc. In formal terms, a row is called a tuple, a column header is called an attribute, and a table is called a relation. In rest of this thesis, we will use the terms relation and table interchangeably. Table 2.1 shows an example

of a PROFESSOR table that stores information about professors at a university. The table has four columns: pnum, pname, office, and deptcode. Each row of this table records information about one professor, consisting of a number that uniquely identifies him in the university, his name, his office, and a code represents the department he belongs to. For instance, the number 2 professor in this university is called "Lucius Lois", works at office "MC_1536", and is part of the "MATH" department.

Table 2.1: The Relational Model: Table Example (PROFESSOR)

pnum	pname	office	deptcode
1	Darrell Stacia	DC_2020	CS
2	Lucius Lois	MC_1536	MATH
3	Carla Valorie	DC_2211	CS
4	Erskine Karin	E5_2567	ECE
5	Primula Timothy	DC_3267	CS

The relational model has some unique characteristics and enforces many constraints. For example, the data type describing the types of values for each column in a table is presented by a domain D. A domain contains all possible and indivisible values for a given attribute. In our PROFESSOR table, the domain of office are all offices located within the university. A data type is also specified for each domain. For instance, the data type of office is the set of all character strings that represent valid office names. Relations also possess some other characteristics: tuples in a relation do not have any particular order, but ordering of attribute values is restricted and important. One may refer to Ramez's book [8] for a more thorough description and explanation of the common characteristics. But in this thesis, we only focus on the constraints imposed by the relational model.

There are in general many different types of constraints in a relational database. Some of them are inherent in the data model, some of them are specified in database schemas, and some of them are enforced by the actual application's business rules. Among the three, we are mostly interested in schema-based constraints as they are the ones the abstract relational model is trying to simplify. A relational database schema describes the logical design of a relational database, and schema-based constraints include domain constraints, key constraints, constraints on NULL values, entity integrity constraints, and referential integrity constraints [8]. Since we are only interested in a subset of these constraints, here we give a formal definition of a simplified relational model $\mathcal{RM}$ that only covers the set of features/constraints we are interested in:

Definition 1 ($\mathcal{RM}$)**.** *Let* TAB, AT, *and* CD *be sets of table names, attribute names, and* concrete domains *(data types), respectively. A* $\mathcal{RM}$ *schema* Σ *is a set of* table declarations

of the form

$$\texttt{table}\ T\ \texttt{(}\ A_1\ D_1\texttt{,}\ \ldots\ \texttt{,}\ A_k\ D_k\texttt{,}\ \varphi_1\texttt{,}\ \ldots\ \texttt{,}\ \varphi_\ell\ \texttt{)}$$

where $T \in \mathsf{TAB}$*,* $A_i \in \mathsf{AT}$*,* $D_i \in \mathsf{CD}$*, and* φ_i *are* constraints *attached to the table* T*. We write* $\mathsf{Attrs}(T)$ *to denote all attributes* $\{A_1, \ldots, A_k\}$ *of table* T*, and* $\mathsf{Dom}(A_i)$ *to refer to* D_i*.* □

In this thesis, we assume a constraint φ_i in a table declaration will be one of the two forms in $\mathcal{RM}$:

1. (primary keys) `primary key` $(A_1, \ldots, A_k)$
2. (foreign keys) `foreign key` $(A_1, \ldots, A_k)$ `references` T

On primary keys, recall that a relation should contain a set of tuples. Therefore, all tuples of a relation must be distinct and can be uniquely identified. A candidate key of a relation is a minimal subset of its attributes that distinguishes any pair of tuples. One designates one of the possible candidate keys to be the primary key of a relation, and its values are then used to identify tuples in this relation. In our `PROFESSOR` table (Table 2.1), the underlined `pnum` is the primary key. Underlining is the commonly used notation to visualize primary keys in tables. A set of entity integrity constraints enforce requirements on the properties of primary keys, such as NULL values are not allowed for primary keys.

Table 2.2: The Relational Model: Table Example (`DEPARTMENT`)

deptcode	deptname
CS	Computer Science
MATH	Mathematics
ECE	Electrical and Computer Engineering

And on foreign keys, recall that many-to-one relationship between a pair of tuples n and s are usually encoded by adding additional attributes to table N, and then adding a foreign key constrain from the added columns to the primary key of S. For example, `deptcode` in our `PROFESSOR` table is a foreign key referring to departments stored in the `DEPARTMENT` table (shown in Table 2.2), where `deptcode` is its primary key. In this case, `PROFESSOR` table is called the referencing relation and `DEPARTMENT` is called the referenced relation of the foreign key. Referential integrity constraint enforces the consistency of a foreign key, by stating that it must match the primary key of another table, and a tuple in one relation can only refer to an existing tuple in another relation. Returning to our

PROFESSOR table, one cannot have a professor with deptcode set to "ART" as there is no matching record in the DEPARTMENT table. Note that primary keys and foreign keys can be composite, too, as sometimes a single attribute may not be enough to uniquely identify a tuple in a relation.

With $\mathcal{RM}$, the table declaration of our PROFESSOR table can be expressed as:

```
table PROFESSOR (
    pnum         INT,
    pname        STRING,
    office       STRING,
    deptcode     STRING,
    primary key  (pnum),
    foreign key  (deptcode) references DEPARTMENT
)
```

where we specify INT as the data type for pnum, and STRING for all other attributes.

A database schema including relations, primary keys and foreign keys can also be depicted by a diagram. The most commonly used diagram is the entity-relationship diagram developed by Peter Chen [6], which can be used to express multiple types of constraints. In this thesis, as we are only interested in primary key and foreign key constraints, we use a much simpler diagram called schema diagram to depict tables and relationships instead [14]. Figure 2.1 shows the schema diagram for a university that stores information regarding courses offered and students' enrollments. In a schema diagram, each relation is represented by a box with the corresponding table name on the top. Attributes are listed inside the box and separated into two parts, where the ones on top form the relation's primary key. The data types of the attributes are not shown in the boxes as they are all concrete and oftentimes can be inferred from attribute names. Foreign key dependencies are illustrated as arrows from the foreign key attributes in the referencing relation to the primary key of the referenced relation. Note that the origin of the arrows can span multiple attributes, since a foreign key can be composite. In our university example (Figure 2.1), the primary key of the SCHEDULE table is deptcode, cnum, term, section, day and time, while the only attribute that is not part of the primary key is room. Additionally, deptcode, cnum, term and section is a foreign key from the SCHEDULE table that is referring to the CLASS table, ensuring that every schedule must have an associated class.

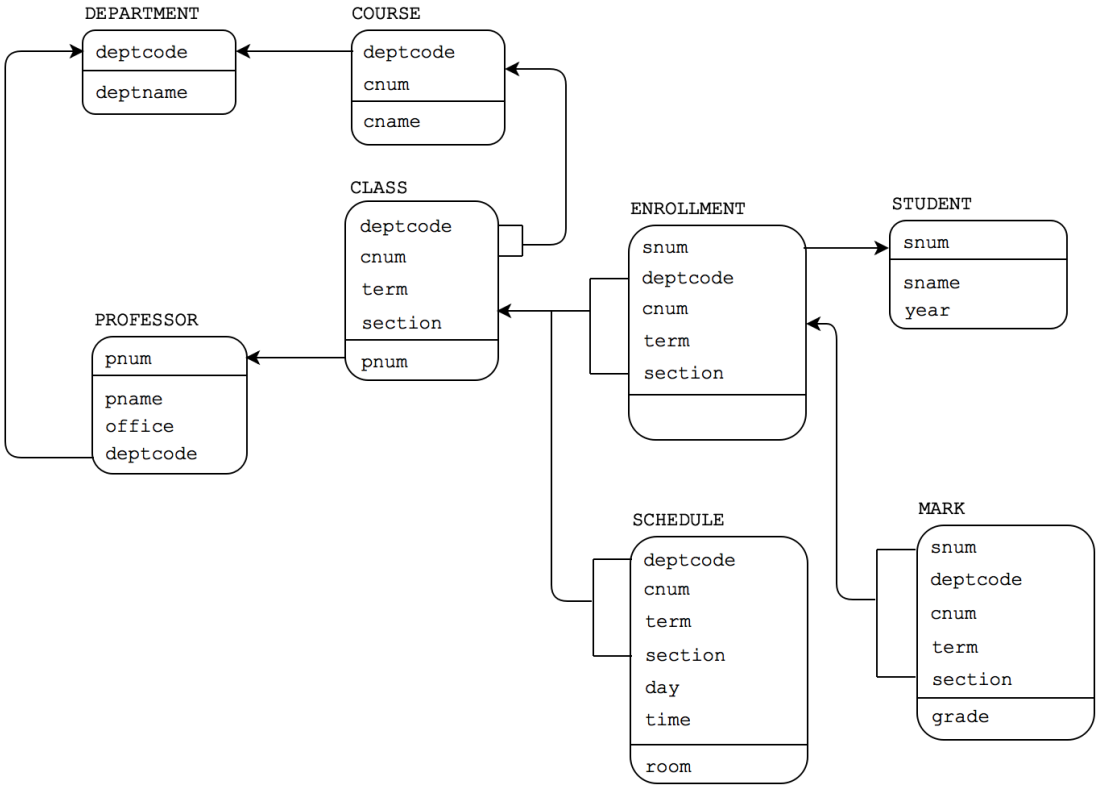

Figure 2.1: Relational Model and Schema for A University

2.2 SQL

In the previous section we discussed how to model and organize data with the relational model in a relational database system. We also need query languages to retrieve stored data from the database. A query language is a high-level programming language that allows users to exchange information with the underlying database system, and can be both procedural or non-procedural. For instance, the relational algebra is a procedural query language, while the tuple and domain relational calculus is a non-procedural query language [7]. Although there are a few implementations of these query languages, they are not adopted by database systems.

Instead, SQL is the standard query language for today's commercial and open source relational databases. SQL is based upon the relational algebra and tuple relational calculus, and was originally developed by IBM Research as the interface to their experimental

relational database system R [5]. It was then standardized by American National Standards Institute (ANSI) and Organization for Standardization (ISO) in 1986 [1]. The SQL standard has evolved over the years and the latest version was published in 2016 [2]. SQL is a comprehensive database query language and contains multiple parts. It is both a Data Definition Language (DDL) and a Data Manipulation Language (DML), thus not only provides interfaces for relation schema definition, deletion, and modification, but also gives users the ability to query, insert, delete, and modify tuples from and to the database. We only cover the basics of the DML part, and more specifically SQL queries here, as we mainly focus on conjunctive queries in this thesis. One may refer to Elmasri's book for more details regarding other parts of SQL [8].

A SQL query is therefore the means to retrieve stored information from the underlying tables. In this thesis, we assume such a query adheres to a basic `select-from-where` form given as follows:

```
select distinct <results>
from <tables>
where <condition>
```

There are three clauses: `select`, `from`, and `where`. One specifies the input relations in the `from` clause, conditions in the `where` clause, and the result columns in the `select` clause. We assume the `distinct` keyword is used in every `select` clause to ensure the resulting table is a set of tuples without duplication. One can apply advanced operations in the `select` clause, such as mathematical computation, to manipulate the result relation. Additionally, complex boolean expressions can be specified in the `where` clause as a predicate to only select satisfying records from the result relation. Take our `PROFESSOR` table for example, "the names of all professors that are from the computer science department" can be specified with the following query:

```
select distinct p.pname
from professor p
where p.deptcode = 'CS'
```

In real world applications, the usage of SQL targeting a single relation is very limited. Oftentimes, we must access information from multiple relations. For instance, we may only know the full name of the "Computer Science" department, but not its `deptcode`. In this case, to retrieve the same result as before, we must match the tuples in the `PROFESSOR` table with the tuples in the `DEPARTMENT` table, taking advantage of the foreign key reference

between them. This can be done by supplying multiple relations to the `from` clause of the query statement, which by itself defines a Cartesian product of the relations listed in the clause. By default, when multiple relations are present in the `from` clause, the result relation will contain all attributes from these relations. We can assign relations aliases and use them as prefixes to distinguish the same attribute names in different relations. For example, if we assign aliases "p" and "d" to `PROFESSOR` and `DEPARTMENT` tables, we can then use "p.deptcode" and "d.deptcode" to distinguish the `deptcode` column in them.

The Cartesian product is a special case of a table join, in particular an inner join. It has the property that all tuples from all provided relations are matched, even if they are unrelated to each other. Take our `PROFESSOR` and `DEPARTMENT` tables for example, each tuple in `PROFESSOR` is combined with every tuple in `DEPARTMENT`, even if one professor only belongs to one department. This will produce a very large result relation, where the majority of its information is useless. We can then use the predicate in the `where` clause to filter out the matched tuples that we are not interested in to overcome this problem. In this example, we only want the join to combine tuples in `PROFESSOR` and `DEPARTMENT` tables that have the same `deptcode`. The following SQL query ensures this requirement, and generate a result relation containing the names of the professors that belong to the "Computer Science" department:

```
select distinct p.pname
from professor p, department d
where p.deptcode = d.deptcode
and d.deptname = 'Computer Science'
```

Note that we used aliases in the above query, and the first boolean expression in the `where` clause is the predicate that filters out tuples that are not interesting to us. This type of SQL query is often called the conjunctive query. Here we present a simplified version of the SQL query syntax that covers everything we are interested in:

Definition 2 (SQL_{query})**.** *The following grammar gives the syntax for a simplified version*

of SQL query language over instances of $\mathcal{RM}$ schema:

$$\begin{aligned}
\langle SQL_{query} \rangle &::= \texttt{select distinct } t_1, t_2, \ldots, t_m \\
&\quad \langle SQL_{body} \rangle \\
\\
\langle SQL_{body} \rangle &::= \texttt{from } T_1\ x_1, T_2\ x_2, \ldots, T_n\ x_n \\
&\quad \texttt{where } \langle cond \rangle \\
\\
\langle cond \rangle &::= t\ \texttt{op}\ t' \\
&\quad \mid t\ \texttt{op}\ c \\
&\quad \mid \langle cond \rangle\ \texttt{and}\ \langle cond \rangle \\
&\quad \mid \texttt{exists (select } *\ \langle SQL_{body} \rangle) \\
&\quad \mid \texttt{not } \langle cond \rangle
\end{aligned}$$

where T_i is a table name, x_i is a variable name, c is a constant, and `op` *represents a conditional operator. t_i, t and t' are terms and they must have the form "$x.A$", such that "$T\ x$" occurs in a containing* `from` *clause and $A \in \mathsf{Attrs}(T)$.* □

Note that the list of terms $t_1, t_2, \ldots, t_m$ in the `select` clause can also be replaced by $*$ to select all attributes presented in the result relation. Additionally, we relaxed the definition to allow the use of `not` operator in the query. Note that allowing `not` in a `where` clause deviates from the conjunctive fragment of SQL's query language, and is necessary in our study as a consequence of one of our test cases. Additionally, we support the inequality operator <> as a conditional operator since it can be translated to equality operator = with the `not` operator. With this syntax, multiple tables can be included in the `from` clause to produce the Cartesian product, and nested queries with `exists` keyword can be included in the `where` clause to answer more complex questions.

Conceptually speaking, a conjunctive query first creates a Cartesian product of the relations provided in the `from` clause, then apply the boolean expressions supplied in the `where` clause as predicates to filter matched tuples in result relation, and output the attributes specified in the `select` clause from the result relation. These steps can help one to understand the idea of conjunctive queries better, but the actual execution plan in database engine is actually very different and highly optimized in real RDBMS.

Although we only cover one of the simplest forms of SQL queries that can be expressed over a database, it is powerful enough to answer many interesting questions. Given the relational schema in Figure 2.1, consider the following SQL query to find all names of professors in the department "Computer Science" that have taught the class "CS348":

```
select distinct p.pname
from class cl, professor p, department d
where cl.pnum = p.pnum
and p.deptcode = d.deptcode
and cl.cnum = 348
and cl.deptcode = 'CS'
and d.deptname = 'Computer Science'
```

this time it joins three tables to provide the result relation we want. To solve problems that are even more complex, we will notice that the number of boolean expressions within the `where` grows as the number of joining tables and arity of foreign key attributes grow. This will make such a SQL query extremely verbose and error prone. In next chapter, we will introduce the abstract relational model, together with referring expression types and SQL^{path}. They can not only simplify the syntax of SQL queries, but also help to resolve other identification issues present in the relational model.

Chapter 3

The Abstract Relational Model

In Borgida, Toman, and Weddell's work, they discovered that entity relationship modellers oftentimes have trouble and have to pay premature attention to naming and identification issues when they are deriving relational database schemata from conceptual models [3]. These problems have also been investigated by another researcher Richard Hull [10]. Although object-centered modelling approaches can mitigate some of the observed problems, the naming issues introduced by inheritance and generalization when heterogeneous subclasses are present and weak entities still remain unhandled by the relational model and SQL.

To resolve such issues, Borgida, Toman, and Weddell first introduced a conceptual modelling schema that extends traditional relational schema by adding abstract domain of identifiers/surrogates. In this chapter, we first present and explain the definition and properties of this conceptual schema with abstract domain. Then we demonstrate what referring expression type is and how to map the abstract relational model back to the relational model with the help of referring expression type assignment. In the end, we present SQL$^{\text{path}}$, an extension to standard SQL, which supports the usage of abstract domain attribute and attribute path. We will illustrate how SQL$^{\text{path}}$ and attribute path can be used to simplify the syntax of complex conjunctive queries.

3.1 An Abstract Domain

The abstract relational model proposed by Borgida, Toman, and Weddell is built on top of the conceptual modelling language $\mathcal{C}_{AR}$ [3]. $\mathcal{C}_{AR}$ supports data declaration and manip-

ulation, and has a syntax very close to SQL. In this thesis, we use a simplified version of $\mathcal{C}_{AR}$ to define the abstract relational model $\mathcal{ARM}$:

Definition 3 ($\mathcal{ARM}$)**.** *Let* TAB*,* AT*, and* CD *be sets of table names, attribute names, and* concrete domains *(data types), respectively, and let* OID *be an* abstract domain *of* identifiers/surrogates*, disjoint from all concrete domains. An $\mathcal{ARM}$ schema Σ is a set of* abstract table declarations *of the form*

$$\texttt{table}\ T\ \texttt{(}\ \texttt{self OID,}\ A_1\ D_1\texttt{,}\ \ldots\ \texttt{,}\ A_k\ D_k\texttt{,}\ \varphi_1\texttt{,}\ \ldots\ \texttt{,}\ \varphi_\ell\ \texttt{)}$$

where $T \in$ TAB*,* self $\in$ OID *is the* primary key *of* T *(*self *is a* distinguished attribute *identifying the aggregation* $(A_1, \ldots, A_k \in \mathsf{AT})$*),* $A_i \in$ AT*,* $D_i \in \mathsf{CD} \cup \{\texttt{OID}\}$*, and* φ_i *are* constraints *attached to the abstract table* T*. We write* $\mathsf{Attrs}(T)$ *to denote all attributes* $\{A_1, \ldots, A_k\}$ *of abstract table* T*, and* $\mathsf{Dom}(A_i)$ *to refer to* D_i*.* □

A constraint φ_i in an abstract table declaration can be one of the five forms:

1. (path functional dependencies) $\texttt{pathfd}\ \texttt{Pf}_1, \ldots, \texttt{Pf}_\texttt{n} \rightarrow \texttt{Pf}$
2. (foreign keys) $\texttt{foreign key}\ (A)\ \texttt{references}\ T$
3. (specialization) $\texttt{isa}\ T$
4. (cover constraints) $\texttt{covered by}\ \{T_1, \ldots, T_m\}$
5. (disjointness constraints) $\texttt{disjoint with}\ \{T_1, \ldots, T_m\}$

Primary keys constraint is not specified here since in $\mathcal{ARM}$, self is the only primary key for all the relations.

$\mathcal{ARM}$ with its basis in logic also gives users the ability to reason about constraints in abstract table declaration as logical consequences. For an abstract table T in abstract schema Σ, we can write $\Sigma \models (\varphi \in T)$ to denote the fact that a particular constraint φ for T (may not be explicit) *logically follows* from the constraints in schema Σ. The problem of deciding when $\Sigma \models (\varphi \in T)$ holds can be reduced to reasoning about logical consequence in the *description logic* $\mathcal{DLFD}$ and $\mathcal{CFD}^{\forall}_{nc}$. Details of such logic reasoning can be found in Toman and Weddell's work [15] [17].

In $\mathcal{ARM}$, the definition of TAB and AT are the same as they are in the relational model. Besides the concrete domains (CD) defined in the relational model, $\mathcal{ARM}$ also introduced the concept of abstract attributes of the abstract domain. The abstract domain is an

abstraction of the actual data types, and is only present in an abstract table, where abstract attributes associated to the abstract domain are not actual columns in the underlying relational database tables, nor any of their values are actually stored.

In this thesis, we mainly focus on path functional dependencies and foreign keys constraints. Instead of using primary key constraints, abstract table declaration allows programmers to define abstract tables with a special abstract primary key `self` of the abstract data type `OID`. The user visible object identifier `OID` is a set of abstract identifiers/surrogates, which is an abstraction of all primary keys in the corresponding relational database schema, and can also be used as foreign key references.

Note that in an abstract table T, `self` is the only attribute forming its primary key, thus there may be no one-to-one mapping between `self` and concrete attributes in the relational model. We can then use the path functional dependencies constraint `pathfd` of T to find those concrete attributes that can form the primary key of T's corresponding relational table. Inside `pathfd`, we assume `Pf` is always `self` in this thesis and each of its component $\texttt{Pf}_\texttt{i}$ is an attribute path that is either a concrete attribute of T or a path of the form $A_1.A_2.\ldots.A_n$ that refers to a concrete attribute A_n in another table following foreign key constraints. More formally, $\texttt{Pf}_\texttt{i} \in \texttt{PF}_{wf}(T)$, where $\texttt{PF}_{wf}(T)$ denotes the set of well-formed attribute paths for table T, and its definition can be found later in the SQL$^{\text{path}}$ section. For foreign keys, an attribute A_i used in a foreign key constraint must be abstract, and match the abstract primary key `self` in another table.

Now with $\mathcal{ARM}$, the abstract table declaration of our previous `PROFESSOR` table can be expressed as:

```
table PROFESSOR (
    self        OID,
    pnum        INT,
    pname       STRING,
    office      STRING,
    department  OID,
    foreign key (department) references DEPARTMENT,
    pathfd pnum → self
)
```

Note that the primary key and foreign key has been replaced by `self` and `department` from the abstract domain `OID`. The path functional dependency constraint `pathfd` asserts that any pair of `PROFESSOR` tuples agreeing on `pnum` also agree on `self`. This can be used

to figure out the primary key of the corresponding relational table, as `pnum` can be used to uniquely identify professors.

For tables with composite primary key in the relational model, such as `COURSE` in the university example (Figure 2.1), its abstract table declaration can be expressed as:

```
table COURSE (
    self        OID,
    cnum        INT,
    cname       STRING,
    department  OID,
    foreign key (department) references DEPARTMENT,
    pathfd cnum, department.deptcode → self
)
```

In this case, foreign key constraint asserts that each value of `department` must appear as the `self` value of a `DEPARTMENT` tuple. Moreover, the path functional dependency `pathfd` asserts that the course number `cnum` and the course's associated department's `deptcode` form the primary key of the `COURSE` relation. Note that here we used attribute path "department.deptcode" in `pathfd`, since `deptcode` is only defined in the `DEPARTMENT` table not the `COURSE` table.

The abstract relational model also has support to generalization relations and sub-relations. If we add a generalization relation `person` that contains all students and professors. Its abstract table declaration can be expressed as:

```
table PERSON (
    self        OID,
    role        enum{'STUDENT', 'PROFESSOR'},
    snum        INT,
    pnum        INT,
    covered by  {STUDENT, PROFESSOR},
    foreign key (department) references DEPARTMENT,
    pathfd snum, pnum → self
)
```

where the cover constraint makes sure that the generalization relation `PERSON` can be covered by its heterogeneous sub-relations `STUDENT` and `PROFESSOR`. Additionally, we need to

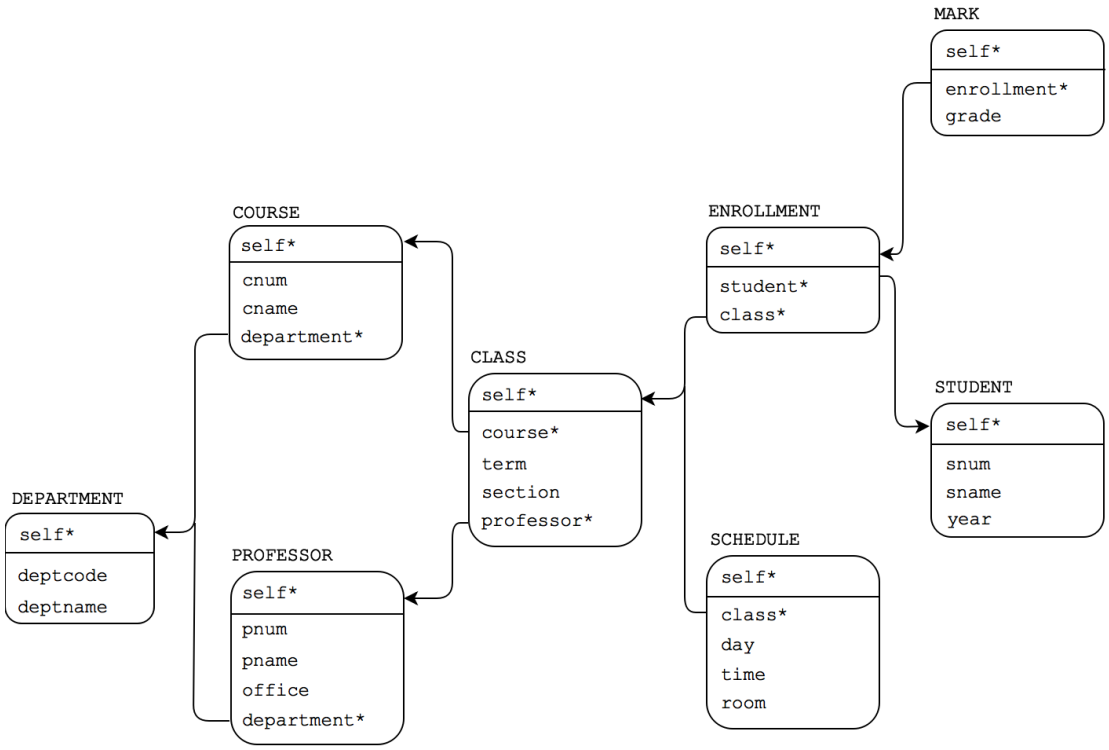

Figure 3.1: Abstract Relational Model and Schema for A University

add specialization constraint and disjointness constraint to our abstract relation `PROFESSOR` and `STUDENT`. We only include the abstract table declaration for `PROFESSOR` here as `STUDENT` would be very similar.

```
table PROFESSOR (
    self        OID,
    pnum        INT,
    pname       STRING,
    office      STRING,
    department  OID,
    foreign key (department) references DEPARTMENT,
    isa PERSON,
    disjoint with {STUDENT},
    pathfd pnum → self
)
```

where the specialization constraint asserts that `self` values of `PROFESSOR` tuples are a subset of `self` values of `PERSON` tuples, and are disjoint from `self` values of `STUDENT` tuples, since a person at the university is either a student or a professor, but not both.

Furthermore, like in the relational model, the abstract relational models and their associated relationships can also be depicted using an abstract schema diagram. For example, the relational schema of the university shown previously in Figure 2.1 can be depicted with the abstract schema diagram and the new diagram is shown in Figure 3.1. In an abstract schema diagram, we use boxes to represent abstract relations, with their names placed on top of the boxes. Primary key attributes and other attributes are still separated. But in this case, the primary key is always `self` for all abstract relations, and all of the abstract attributes of `OID` are marked by "*".

3.2 Referring Expression Types

By its design, $\mathcal{ARM}$ provides separation of concerns between identification issues and data access requirements. It allows high level application developers to focus more on the logic and relationships between abstract relations, without worrying much about the actual attributes and how they are used to identify underlying entities. However, it comes with a side effect that the values of some of the attributes are purely abstract and cannot be stored in actual tables in the database. This must be resolved as eventually we must translate the entire database schema from the abstract relational models back to the relational model to make it work with today's RDBMS. In Borgida, Toman, and Weddell's work, this is handled by the *referring expression type language* [4]. The definition of this language is presented below:

Definition 4 (Referring Expressions, Types, and Assignments)**.** *Let* Σ *be a* $\mathcal{ARM}$ *schema. A* referring expression type Rt relative to Σ *is an instance of a recursive* pattern *language given by the grammar:*

$$Rt \quad ::= \quad \texttt{Pf} = ? \;\mid\; Rt, Rt \;\mid\; G \to Rt, \;\mid\; Rt; Rt$$

where `Pf` *is an attribute path ending in a concrete attribute, and where* $G = \{T_1, \ldots, T_\ell\}$ *is a set of table names from* $\mathsf{Tables}(\Sigma)$*, called a* guard*. We write* $\mathrm{RE}(Rt)$ *to refer to a set of* referring expressions ϕ_i *induced by a given referring expression type* Rt *relative to* Σ *as*

follows:

$$\begin{aligned} R_E(\texttt{Pf} = ?) &= \{x.\texttt{Pf} = a \mid a \text{ a constant}\} \\ R_E(Rt_1, Rt_2) &= \{\phi_1 \wedge \phi_2 \mid \phi_i \in R_E(Rt_i)\} \\ R_E(\{T_1, \ldots, T_k\} \rightarrow Rt) &= \{\textstyle\bigwedge_{i=1}^{k}(\exists y_1, \ldots, y_l.T_i(x, y_1, \ldots, y_l)) \wedge \phi \mid \phi \in R_E(Rt)\} \\ R_E(Rt_1; Rt_2) &= R_E(Rt_1) \cup \{\phi \in R_E(Rt_2) \mid \neg\exists\psi \in R_E(Rt_1).(\phi \equiv \psi)\} \end{aligned}$$

Given $T \in \mathsf{Tables}(\Sigma)$*, we say that* Rt *is* strongly identifying *for* T *if, for all instances* I *of* Σ*,*

$$\begin{aligned} I \models \forall x_1, x_2.(&\exists y_1, \ldots, y_l.T(x_1, y_1, \ldots, y_l) \wedge \phi(x/x_1)) \wedge \\ &(\exists y_1, \ldots, y_l.T(x_2, y_1, \ldots, y_l) \wedge \phi(x/x_2)) \rightarrow x_1 = x_2, \end{aligned}$$

holds for all $\phi \in R_E(Rt)$*, and*

$$I \models \neg\exists x.(\phi_1 \wedge \phi_2)$$

holds for all syntactically distinct $\phi_1, \phi_2 \in R_E(Rt)$*.*

A referring type assignment *for* Σ *is a mapping* RTA *from* $\mathsf{Tables}(\Sigma)$ *to referring expression types relative to* Σ*.* □

One can refer to Borgida, Toman, and Weddell's papers [3] [4] for a more detailed explanation of referring expression types. Overall, the referring expression type assigned by RTA to an abstract table T, $\mathsf{RTA}(T)$, provides us the attribute paths leading to concrete attributes that can be used to identify tuples in table T. These attributes can be included in the corresponding relational table to form its key. Generally speaking, we want to find a RTA that can strongly identify all of our declared abstract tables and satisfy all of the constraints in our abstract relational schema. A referring expression type, Rt has a normal form given as follows [4] [3]:

$$T_1 \rightarrow (\texttt{Pf}_{1,1} = ?, \ldots, \texttt{Pf}_{1,k_1} = ?); \ldots; T_k \rightarrow (\texttt{Pf}_{k,1} = ?, \ldots, \texttt{Pf}_{k,k_k} = ?)$$

where T_i is a table name and where each $\texttt{Pf}_{i,j}$ is a well-formed attribute path on table T_i. When specialization constraints are not present and all tables are disjoint, we can further simplify $\mathsf{RTA}(T)$ for any T to the form:

$$T \rightarrow (\texttt{Pf}_1 = ?, \ldots, \texttt{Pf}_k = ?)$$

To illustrate with our example in Figure 3.1, $\mathsf{RTA}(\texttt{PROFESSOR})$ can be assigned as "`PROFESSOR` $\rightarrow$ (`pnum` =?)" (where `pnum` strongly identifies professors). Also, since referring expression types Rt admit attribute paths as components (when foreign key constraints are present), $\mathsf{RTA}(\texttt{COURSE})$ would be "`COURSE` $\rightarrow$ (`cnum` =?, `department.deptcode` =?)".

Thus, a referring expression type assignment RTA may not be unique, and a Rt can contain attribute paths that are not necessary. In Borgida, Toman, and Weddell's paper [3], they provided a way to handle this with non-redundant referring types and identify resolving type assignments, by introducing referring expression $\mathsf{Fix}(Rt, T)$ and $\mathsf{Prune}(Rt, T)$ that can be used to remove redundant referring types. In addition, they give a formal definition of an identity resolving referring expression type assignment by introducing a linear ordering of all tables in Σ such that when guards in RTA are in this exact order all attribute paths in each component of the RTA are well defined and all tables are strongly identified. We refer the readers to the reference mentioned early for further details.

In the next section, we will present a systematic approach to map the relational model and database schema to the abstract relational model and schema, as well as produce the associated strongly identifying RTA that assign referring expression types to all abstract tables. Moreover, we will demonstrate how to convert the abstract relational model and schema with its RTA back to the relational model and schema, thus one can derive the relational tables that can be created and queried in real RDBMS.

3.3 Mapping between Relational Model and Abstract Relational Model

From our definition of the abstract relational model in previous sections, an abstract relation has two major differences when compared with its relational counterpart. First, its primary key is no longer a concrete attribute, and becomes the user visible object identifier `self`. Second, since a concrete attribute can only be declared in at most one abstract table, all attributes that are part of foreign key constraints referring to another table's primary key in the original relational table are compressed and replaced by an `OID` attribute that is used in the updated foreign key constraint. The replaced concrete attributes are only declared in tables where they were originally defined as the primary key and not part of any foreign key constraints.

Based on these properties, we present a procedure to convert a relational schema Σ to a counterpart abstract relational schema together with a valid RTA that can strongly identify all tuples. To perform this conversion, we need to first create a *pending assignment* set PA and a stack of tables S. A 4-tuple $(T, \mathtt{Pf}, T', A) \in \mathsf{PA}$ states that the primary key of T has component $\mathtt{Pf} \circ A$, and depends via foreign key join path `Pf`, on table T' having

attribute A. We also introduce the composition operator $\circ$ such that:

$$\texttt{Pf}_1 \circ \texttt{Pf}_2 \equiv \begin{cases} \texttt{Pf}_1, & \text{if } \texttt{Pf}_2 \text{ is } \texttt{self} \\ \texttt{Pf}_2, & \text{if } \texttt{Pf}_1 \text{ is } \texttt{self} \\ \texttt{Pf}_1.\texttt{Pf}_2, & \text{otherwise} \end{cases}$$

The procedure contains two parts where we initialize PA, S, and RTA first before generating abstract tables.

(Initialization)

1. Assign $\mathsf{PA} \leftarrow \emptyset$, $\mathsf{S} \leftarrow []$.
2. For each $T \in \mathsf{Tables}(\Sigma)$ with primary key $(A_1, \ldots, A_m)$:
   2.1. Assign $\mathsf{RTA}(T)$ as $T \rightarrow (A_1 = ?, \ldots, A_m = ?)$.
   2.2. Add $(T, \texttt{self}, T, A_i)$ to PA for $1 \leq i \leq m$.
3. Construct a directed graph $G(\mathsf{Tables}(\Sigma), E)$, where E is obtained as follows: for each $T_1 \in \mathsf{Tables}(\Sigma)$ with primary key $(A_1, \ldots, A_m)$, iterate through each foreign key constraint φ for T_1 of the form "`foreign key` $(B_1, \ldots, B_n)$ `references` T_2". When $\{A_1, \ldots, A_m\} \cap \{B_1, \ldots, B_n\}$ is not empty and T_1 is not reachable from T_2 in G, add $T_1 \rightarrow T_2$ to E.
4. While there exists $T \in V_G$ where T's *outdegree* is 0:
   4.1. Push T on S.
   4.2. Remove T from G together with incident edges.

($\mathcal{ARM}$ Generation)

5. While S is not empty, do the following:
   5.1. Pop T from S, where the primary key of T is $(A_1, \ldots, A_m)$.
   5.2. Add `self` to $\mathsf{Attrs}(T)$, where $\mathsf{Dom}(\texttt{self}) = \texttt{OID}$.
   5.3. Find a list of tables $T_1, \ldots, T_n \in \mathsf{Tables}(\Sigma)$ where T_i's primary key is $(A_{i,1}, \ldots, A_{i,k})$ and $\{A_{i,1}, \ldots, A_{i,k}\} \neq \{A_1, \ldots, A_m\}$. If this list is not empty, add disjointness constraint "`disjoint with` $\{T_1, \ldots, T_n\}$" to T.

5.4. For each foreign key constraint $\varphi =$ `foreign key` $(B_1, \ldots, B_k)$ `references` T' defined for T where the primary key of T' is $(C_1, \ldots, C_k)$, do the following:

5.4.1. If $\{B_1, \ldots, B_k\} = \{A_1, \ldots, A_k\}$, then replace φ by specialization constraint "`isa` T'", assign $\mathsf{NA} \leftarrow$ `self`, and proceed to 5.4.4.

5.4.2. When attributes $(A_1, \ldots, A_k)$ and $(B_1, \ldots, B_k)$ are exactly the same, or φ is the only foreign key constraint of T that references T', then assign $\mathsf{NA} \leftarrow \mathsf{name}(T')$, where $\mathsf{name}(T')$ retrieves T''s table name in lower case. Otherwise, assign $\mathsf{NA} \leftarrow B_1\text{-}\ldots\text{-}B_k\text{-}ref$. Add NA to $\mathsf{Attrs}(T)$, where $\mathsf{Dom}(\mathsf{NA}) =$ `OID`.

5.4.3. Replace φ by foreign key constraint "`foreign key` (NA) `references` T'".

5.4.4. For each B_i, $1 \leq i \leq k$, if $C_i \in \mathsf{Attrs}(T')$ and if all remaining foreign key constraints φ' for T are free of B_i, then do the following:

- Remove B_i from $\mathsf{Attrs}(T)$.
- If $B_i \in \{A_1, \ldots, A_k\}$, for every tuple $t \in \mathsf{PA}$ of the form $(T'', \mathtt{Pf}, T, B_i)$, for some T'' and `Pf`:
  Replace "$\mathtt{Pf} \circ B_i = ?$" in $\mathsf{RTA}(T'')$ by "$\mathtt{Pf} \circ \mathsf{NA} \circ C_i = ?$", and replace t itself by $(T'', \mathtt{Pf} \circ \mathsf{NA}, T', C_i)$.

6. For each $T \in \mathsf{Tables}(\Sigma)$, replace the primary key constraint φ in T by path functional dependencies "`pathfd` $\mathtt{Pf}_1, \ldots, \mathtt{Pf}_m \rightarrow$ `self`", where $\mathsf{RTA}(T) = T \rightarrow (\mathtt{Pf}_1 = ?, \ldots, \mathtt{Pf}_m = ?)$.

Going back to our university example and its relational database schema diagram depicted in Figure 2.1, we can follow the steps above to map it to the abstract relational table declarations with RTA. Take the relation `COURSE` for instance, after initialization, its related tuples in PA are (`COURSE`, `self`, `COURSE`, `deptcode`) and (`COURSE`, `self`, `COURSE`, `cnum`). When popped out for processing in step 5.1, its table declaration is:

```
table COURSE (
    deptcode    STRING,
    cnum        INT,
    cname       STRING,
    primary key (deptcode, cnum),
    foreign key (deptcode) references DEPARTMENT
)
```

After step 5.2 and 5.3, it becomes:

```
table COURSE (
    self       OID,
    deptcode   STRING,
    cnum       INT,
    cname      STRING,
    disjoint with {DEPARTMENT, PROFESSOR, CLASS,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    primary key (deptcode, cnum),
    foreign key (deptcode) references DEPARTMENT
)
```

In step 5.4.2 and 5.4.3, we add abstract attribute `department` since `deptcode` matches `DEPARTMENT`'s primary key. Also add its associated foreign key constraint to replace the original foreign key constraint of `deptcode` in `COURSE`:

```
table COURSE (
    self         OID,
    deptcode     STRING,
    cnum         INT,
    cname        STRING,
    department   OID,
    disjoint with {DEPARTMENT, PROFESSOR, CLASS,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    primary key (deptcode, cnum),
    foreign key (department) references DEPARTMENT
)
```

In step 5.4.4, the concrete attribute `deptcode` is removed:

```
table COURSE (
    self         OID,
    cnum         INT,
    cname        STRING,
    department   OID,
    disjoint with {DEPARTMENT, PROFESSOR, CLASS,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    primary key (deptcode, cnum),
    foreign key (department) references DEPARTMENT
)
```

Tuple (COURSE, self, COURSE, deptcode) ∈ PA is replaced by (COURSE, department, DEPARTMENT, deptcode), while RTA(COURSE) is updated to

$$\mathsf{RTA}(\texttt{COURSE}) = \texttt{COURSE} \rightarrow (\texttt{department.deptcode} = ?, \texttt{cnum} = ?)$$

as the concrete attribute deptcode for its primary key was replaced by OID department. In the end, we add the path functional dependency based on the final version of RTA(COURSE) to COURSE to replace its primary key constraint and complete its declaration:

```
table COURSE (
    self         OID,
    cnum         INT,
    cname        STRING,
    department   OID,
    disjoint with {DEPARTMENT, PROFESSOR, CLASS,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    foreign key (department) references DEPARTMENT,
    pathfd cnum, department.deptcode → self
)
```

A more complex example is the CLASS relation, which based on our procedure is processed right before COURSE. Its related tuples in PA after initialization are (CLASS, self, CLASS, deptcode), (CLASS, self, CLASS, cnum), (CLASS, self, CLASS, term), and (CLASS, self, CLASS, section). After step 5.3, its table declaration is:

```
table CLASS (
    self       OID,
    deptcode   STRING,
    cnum       INT,
    term       STRING,
    section    INT,
    pnum       INT,
    disjoint with {DEPARTMENT, PROFESSOR, COURSE,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    primary key (deptcode, cnum, term, section),
    foreign key (deptcode, cnum) references COURSE,
    foreign key (pnum) references PROFESSOR
)
```

Since there are two forgien key constraints, step 5.4 is iterated twice. In steps 5.4.2 and 5.4.3, the abstract attributes `course` and `professor` are added in the two iterations since the attributes of the foreign keys are exactly the same as `COURSE`'s and `PROFESSOR`'s primary key. In addition, the two foreign key constraints are replaced:

```
table CLASS (
    self        OID,
    course      OID,
    deptcode    STRING,
    cnum        INT,
    term        STRING,
    section     INT,
    pnum        INT,
    professor   OID,
    disjoint with {DEPARTMENT, PROFESSOR, COURSE,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    primary key (deptcode, cnum, term, section),
    foreign key (course) references COURSE,
    foreign key (professor) references PROFESSOR
)
```

Then in 5.4.4, concrete attributes `deptcode`, `cnum`, and `pnum` are removed, and tuples (`CLASS`, `self`, `CLASS`, `deptcode`) and (`CLASS`, `self`, `CLASS`, `cnum`) $\in$ PA are replaced by (`CLAS S`, `course`, `COURSE`, `deptcode`), and (`CLASS`, `course`, `COURSE`, `cnum`). Note that these tuples may be further updated again when `COURSE` table is processed. Recall step 5.4.4 for the `COURSE` table, after its own tuples being replaced, the first tuple for `CLASS` will eventually be replaced by (`CLASS`, `course.department`, `DEPARTMENT`, `deptcode`). Therefore, RTA(`CLASS`) in the end is updated to:

$$\text{RTA}(\texttt{CLASS}) = \texttt{CLASS} \rightarrow \ (\texttt{course.department.deptcode} = ?, \\ \texttt{course.cnum} = ?, \texttt{term} = ?, \texttt{section} = ?)$$

When everything is done, the path functional dependency based on RTA(`CLASS`) is added to `CLASS` to replace its primary key and the completed abstract table declaration becomes:

```
table CLASS (
    self        OID,
    course      OID,
```

```
    term        STRING,
    section     INT,
    professor   OID,
    disjoint with {DEPARTMENT, PROFESSOR, COURSE,
                   ENROLLMENT, SCHEDULE, STUDENT, MARK},
    foreign key (course) references COURSE,
    foreign key (professor) references PROFESSOR,
    pathfd course.department.deptcode,
           course.cnum, term, section → self
)
```

On the other hand, given an abstract relational schema Σ and its associated RTA for $\mathsf{Tables}(\Sigma)$ generated by our previous procedure, translate it back to the schema of its corresponding relational model is much simpler. For each abstract relational table $T \in \mathsf{Tables}(\Sigma)$, do the following:

1. Iterate through all components of $\mathsf{RTA}(T) = T \rightarrow (\mathtt{Pf}_1 = ?, \ldots, \mathtt{Pf}_k = ?)$. For i_{th} component "$\mathtt{Pf}_i = ?$" where $\mathtt{Pf}_i = a_1.\ldots.a_n.A_i$ and A_i is a concrete attribute, add A_i to T if it's not already present.

2. Remove `self` from $\mathsf{Attrs}(T)$.

3. Replace path functional dependencies "$\mathtt{pathfd}\ \mathtt{Pf}_1, \ldots, \mathtt{Pf}_k \rightarrow \mathtt{self}$" by primary key constraint "$\mathtt{primary\ key}\ (A_1, \ldots, A_k)$".

4. For any other concrete attribute in $\mathsf{Attrs}(T)$ that is not part of $\mathsf{RTA}(T)$, keep them unchanged in the relational table (not part of the primary key).

5. For each foreign key constraint $\varphi = \mathtt{foreign\ key}\ A\ \mathtt{references}\ T'$ defined for T, do the following:

   5.1. Iterate through all components of $\mathsf{RTA}(T') = T' \rightarrow (\mathtt{Pf}_1 = ?, \ldots, \mathtt{Pf}_m = ?)$. For i_{th} component $\mathtt{Pf}_i = ?$ where $\mathtt{Pf}_i = a_1.\ldots.a_n.A_i$, if "$A.\mathtt{Pf}_i = ?$" doesn't exist in $\mathsf{RTA}(T)$, add A_i to $\mathsf{Attrs}(T)$ with prefix "A-" to avoid duplicate attribute names.

   5.2. Add foreign key constraint "$\mathtt{foreign\ key}\ (A_1, \ldots, A_m)\ \mathtt{references}\ T'$", where A_i has prefix "A-" if it was added to $\mathsf{Attrs}(T)$ in 5.1.

   5.3. Remove φ from T.

6. Remove all abstract attributes from $\mathsf{Attrs}(T)$.

Following these steps, we can map our previously declared abstract table `COURSE` with $\mathsf{RTA}(\texttt{COURSE}) = \texttt{COURSE} \rightarrow (\texttt{department.deptcode} = ?, \texttt{cnum} = ?)$ back to a relational table. After step 1 and 2, the only abstract attribute left is `department`, while its concrete attributes are `deptcode`, `cnum` and `cname`. After step 3, `pathfd` is removed and primary key constraint "`primary key(deptcode, cnum)`" is added. In step 5, there is only one foreign key constraint "`foreign key (department) references DEPARTMENT`" to handle. Since we have $\mathsf{RTA}(\texttt{DEPARTMENT}) = \texttt{DEPARTMENT} \rightarrow (\texttt{deptcode} = ?)$ and the concrete attribute `deptcode` was already added to `COURSE`, we only need to replace the foreign key constraint by "`foreign key (deptcode) references DEPARTMENT`" . In the end, we remove `department` to make sure no abstract attribute exists in the relational table.

With the steps proposed in this section, we can convert our relational schema diagram shown in Figure 2.1 to its abstract relational counterpart and the abstract relational schema diagram shown in Figure 3.1 back to its relational counterpart. Note that these two diagrams are later used in our empirical study so we made some small changes to make them cleaner for experiment participants. For instance, the abstract `MARK` table uses explicit abstract attribute `enrollment` instead of `self` for its foreign key referrencing the `ENROLLMENT` table, and no prefix is used as it's not necessary in this database schema. Therefore, they do not exactly match the resulting schema obtained by applying the steps listed in this section. However, they are semantically equivalent to the results and will not have any side effect on the result of our experiments.

3.4 SQL$^{\text{path}}$

Based on their conceptual modelling language and the abstract relational model, Borgida, Toman, and Weddell also introduced SQL$^{\text{path}}$, an extension to SQL that adopts its core relational algebra fragment and is specifically designed for conjunctive queries [3]. SQL$^{\text{path}}$ incorporates attribute paths and is expected to simplify the syntax for table joins to speed up the composition of complex conjunctive queries, especially for chained foreign key constraints. Their definitions of *attribute path* and SQL$^{\text{path}}$ are presented below:

Definition 5 ($\texttt{PF}_{wf}$: Well-Formed Attribute Path)**.** *Let* $\texttt{PF}_{wf}(T)$ *denotes the set of well-formed attribute paths for table* T*. An attribute path* $\texttt{Pf} \in \texttt{PF}_{wf}(T)$ *if* `Pf` *is* `self` *or, when* `Pf` *has the form* $A \circ \texttt{Pf}'$*, there exists tables* $T_1, \ldots, T_k$ *such that:*

1. $\Sigma \models (\texttt{covered by } \{T_1, \ldots, T_k\}) \in T$,
2. $A \in \mathsf{Attrs}(T_i)$, $1 \leq i \leq k$, *and when* A *is abstract, there exists* T_i', $1 \leq i \leq k$ *such that:*

(a) $\Sigma \models (\texttt{foreign key}\ A\ \texttt{references}\ T_i') \in T_i$, *and*

(b) $\texttt{Pf}' \in \texttt{PF}_{wf}(T_i')$.

Let $\mathsf{Home}(A)$ *denote the set of tables* $T_1, \ldots, T_k$ *for which* $A \in \mathsf{Attrs}(T_i),\ 1 \leq i \leq k$. *When* $\mathsf{Home}(A)$ *is a singular set for* A, *this simplifies to* $\texttt{Pf} \in \texttt{PF}_{wf}(T)$ *if* `Pf` *is* `self`, *or, when* `Pf` *has the form* $A \circ \texttt{Pf}'$, *there exists* T_ℓ, *such that:*

$\Sigma \models (\texttt{covered by}\ \{\mathsf{Home}(A)\}) \in T_\ell$, *and when* A *is abstract, there exists* T_ℓ' *such that:*

(a) $\Sigma \models (\texttt{foreign key}\ A\ \texttt{references}\ T_\ell') \in T_\ell$, *and*

(b) $\texttt{Pf}' \in \texttt{PF}_{wf}(T_\ell')$. □

The composition $A \circ \texttt{Pf}'$ is defined as

$$A \circ \texttt{Pf}' \equiv \begin{cases} A, & \text{if } \texttt{Pf}' \text{ is } \texttt{self} \\ A.\texttt{Pf}', & \text{otherwise} \end{cases}$$

For example, with the abstract relational model depicted in Figure 3.1, "self", "term", "course.cnum", "course.department", and "course.department.deptname" are well-formed attribute paths for the abstract table `CLASS`, while "cname" and "enrollment.student.year" are not since `cname` and `enrollment` are not attributes of `CLASS`.

Definition 6 ($SQL^{\text{path}}{}_{query}$)**.** *The following grammar gives the syntax for (an idealized) SQL-like query language over instances of* $\mathcal{ARM}$ *schema:*

$$\begin{aligned}
\langle SQL^{path}{}_{query} \rangle &::= \texttt{select distinct}\ t_1, t_2, \ldots, t_m \\
&\quad\ \langle SQL^{path}{}_{body} \rangle \\
\\
\langle SQL^{path}{}_{body} \rangle &::= \texttt{from}\ T_1\ x_1, T_2\ x_2, \ldots, T_n\ x_n \\
&\quad\ \texttt{where}\ \langle cond \rangle \\
\\
\langle cond \rangle &::= t\ \texttt{op}\ t' \\
&\quad\ |\ t\ \texttt{op}\ c \\
&\quad\ |\ \langle cond \rangle\ \texttt{and}\ \langle cond \rangle \\
&\quad\ |\ \texttt{exists}\ (\texttt{select}\ *\ \langle SQL^{path}{}_{body} \rangle) \\
&\quad\ |\ \texttt{not}\ \langle cond \rangle
\end{aligned}$$

where T_i *is a table name,* x_i *is a variable name,* c *is a constant, and* `op` *represents a conditional operator.* t_i, t *and* t' *are terms and they must have the form "*x.`Pf` *", such that*

"T x" occurs in a containing `from` *clause and* $\texttt{Pf} \in \texttt{PF}_{wf}(T)$*. Note that all variables are required to be appropriately bounded with a "T x" clause, and denote T by* $\mathsf{Bound}(x)$*, that* `Pf` *is well defined for* $\mathsf{Bound}(x)$ *for any term "x.*`Pf`*".* □

As we can see, this adds a few new features to standard SQL conjunctive queries. First, it allows the use of attribute paths in the `select` and `where` clause, where users are no longer limited to only concrete attributes and can compare attributes of `OID` types for table joins. Second, "T x" is allowed as a independent query, where SQL would require "`select from` T x". It also adds syntactic sugar such as conjunction in `where` clause and multi-arity `from` clauses instead of nested `from` for better readability.

The usage of attribute paths gives users the ability to walk along foreign key constraints and refer to an attribute A_k defined in abstract table $T_k = \mathsf{Home}(A_k)$ from table T_1 with chained attributes $x.A_1., \ldots, .A_k$, given that $T_1 = \mathsf{Bound}(x)$ and there exist abstract tables $T_1, \ldots, T_k$, where A_i is an `OID` attribute such that $\Sigma \models (\texttt{foreign key}\ A_i\ \texttt{references}\ T_{i+1}) \in T_i$, for $1 \leq i \leq k-1$. This can drastically reduce the number of boolean expressions used in the `where` clause to filter result relation for table joins. Recall our example SQL query to find all names of professors in the department "Computer Science" that have taught the class "CS348". Now with SQL$^{\text{path}}$ and the abstract relational model (Figure 3.1), it can be expressed as:

```
select distinct cl.professor.pname
from class cl
where cl.professor.department.deptname = 'Computer Science'
and cl.course.cnum = 348
and cl.course.department.deptname = 'CS'
```

It now has 3 boolean expressions in the `where` clause instead of 5, as we no longer need to explicitly assert the equality of the primary/foreign key attributes in intermediate tables (they are hidden in the attribute paths).

In general, the benefit of SQL$^{\text{path}}$ is more noticeable when the number of tables that need to be joined to answer the query is large, especially when majority of these tables are only used as intermediate steps such that we are only involving them to use their associated foreign key constraints to connect the tables we are actually interested in, rather than using their own attributes. The reduction of boolean expression counts in the `where` clause become even more significant when more attributes are combined to form foreign keys and primary keys of these intermediate tables, as we have to assert equality for every attribute of these keys in SQL. For example, with the relational database schema depicted in Figure

2.1, the SQL conjunctive query to select all professors that are from a department which has ever offered a course that some student has scored a full mark (100) in it is:

```
select distinct p.pnum, p.pname
from professor p
where exists (
        select * from department d
        where d.deptcode = p.deptcode
        and exists (
                select * from course c
                where c.deptcode = d.deptcode
                and exists (
                        select * from mark m
                        where m.grade = 100
                        and m.cnum = c.cnum
                        and m.deptcode = c.deptcode
                )
        )
)
```

Note that although we are only interested in the `PROFESSOR` and `MARK` table, since they are connected by the `DEPARTMENT` and `COURSE` tables, we have to include these two tables in the nested subqueries as well as assert the equality of their primary/foreign keys. The same query, when expressed in SQL$^{\text{path}}$, using the abstract database schema depicted in Figure 3.1, is much simpler:

```
select distinct p.pnum, p.pname
from professor p
where exists (
        select * from mark m
        where m.grade = 100
        and m.enrollment.class.course.department = p.department
)
```

where the usage of attribute path "enrollment.class.course.department" hides the details of intermediate table joins for the abstract tables `ENROLLMENT, CLASS`, and `COURSE`.

Furthermore, the ability to compare `OID` attributes in SQL$^{\text{path}}$ can shorten primary key equality checks, since it's no longer necessary to check its compositing attributes one by one.

For instance, with the university database schema, to check whether two enrollments e_1 and e_2 are of same class, with SQL we have to assert that $\texttt{e}_1\texttt{.deptcode} = \texttt{e}_2\texttt{.deptcode and } \texttt{e}_1\texttt{.cnum} = \texttt{e}_2\texttt{.cnum and } \texttt{e}_1\texttt{.term} = \texttt{e}_2\texttt{.term and } \texttt{e}_1\texttt{.section} = \texttt{e}_2\texttt{.section}$, since the primary key of the `class` table contains four attributes. But with SQL$^{\text{path}}$ the comparison can be simplified to $\texttt{e}_1\texttt{.class} = \texttt{e}_2\texttt{.class}$, as primary keys are replaced by `OID` attributes in abstract tables.

To make SQL$^{\text{path}}$ applicable in real world applications, there must be a systematic way to translate queries written in it back to SQL queries, as today's RDBMS only supports SQL. In Borgida, Toman, and Weddell's work [3], they designed a Map function that can compile any SQL$^{\text{path}}$ query over an abstract schema to a concrete SQL query, given an identity resolving referring expression type assignment RTA for the abstract schema. As the definition of Map is purely syntactic, its behavior is deterministic. Since we can always systematically obtain an RTA when we map relational schema to abstract relational schema as we discussed in previous section, this guarantees that we can obtain concrete SQL queries for all of our SQL$^{\text{path}}$ queries.

The actual definition of the Map function is rather complicated, since it must handle all five supported constraints of the abstract relational model. One may refer to Borgida, Toman, and Weddell's paper for the detailed definition if interested [3]. In general, RTA is used to populate concrete attributes and figure out primary key and foreign key attributes, as we discussed in previous section when translating the abstract relational model back to the corresponding relational model. Then for all attribute paths, tables appear along the paths are used for table joins, while RTA is used to determine primary key and foreign key attributes for equality assertion. Then required attributes from the tables we are interested in are selected for the result relation. For intermediate tables along an attribute path, if its primary key is part of the primary key of a prior table and none of its non-primary key attribute is used, it can be skipped in table joins. For example, the SQL$^{\text{path}}$ query to find all students who have enrolled in class CS341 with the abstract relational database schema in Figure 3.1 is

```
select distinct e.student.snum, e.student.sname
from enrollment e
where e.class.course.department.deptcode = 'CS'
and e.class.course.cnum = 341
```

After applying Map to this query, we obtain $\text{RTA}(\texttt{STUDENT}) = \texttt{STUDENT} \rightarrow (\texttt{snum} = ?)$, $\text{RTA}(\texttt{ENROLLMENT}) = \texttt{ENROLLMENT} \rightarrow (\texttt{student.snum} = ?, \texttt{class.course.cnum} = ?, \texttt{class.course.department.deptcode} = ?, \texttt{class.term} = ?, \texttt{class.section} = ?)$, $\text{RTA}(\texttt{CLASS}) =$

CLASS $\rightarrow$ (course.department.deptcode = ?, course.cnum = ?, term = ?, section = ?), RTA(COURSE) = COURSE $\rightarrow$ (department.deptcode = ?, cnum = ?), and RTA(DEPARTMENT) = DEPARTMENT $\rightarrow$ (deptcode = ?), it can be translated to the following concrete SQL query:

```
select distinct s.snum, s.sname
from student s, enrollment e
where s.snum = e.snum
and e.deptcode = 'CS'
and e.cnum = 341
```

Note that from the given RTA we know the primary key of the concrete `enrollment` table already covers the primary keys of `class`, `course`, and `department` tables. Additionally, we are not interested in any of the non-primary key attributes of these tables. Therefore, we don't need to explicitly join them in the corresponding SQL conjunctive query.

In this chapter, we presented the definitions of the conceptual schema $\mathcal{ARM}$ based on $\mathcal{C}_{AR}$, referring expression type, attribute path, and SQL$^{\text{path}}$ from Borgida, Toman, and Weddell's work. This can be used as both an introduction and tutorial on how to use the abstract relational model and SQL$^{\text{path}}$ to solve conjunctive query problems. We also explained how to map between the abstract relational model and the relational model, with a systematic approach, and demonstrated how to write conjunctive SQL$^{\text{path}}$ queries and convert them back to concrete SQL queries with examples.

However, one may question the actual usability and benefit of SQL$^{\text{path}}$ in real world settings. Although syntactically it can simplify multi-table joins and redundant assertion of composite primary/foreign keys involved in those joins, this may not be appreciated by application developers. To answer this question, we designed and conducted an empirical study at University of Waterloo, and hired students with SQL background to work on real world queries with both SQL and SQL$^{\text{path}}$. The experiment procedure, result, and evaluation of the result data will be presented in next chapter.

Chapter 4

Empirical Study

In last chapter we presented the concept of the abstract relational model and SQL^{path}. To investigate their utility compared to the relational model and SQL, we conducted an empirical study to evaluate their performance when solving real world query problems. We hosted two experiments with undergrad students and graduate students respectively at University of Waterloo. In this Chapter, we will present our hypotheses first, and then explain the detailed experiments' procedures as well as the results with statistical analysis.

4.1 Hypotheses

Recall that SQL^{path} has two potential advantages over traditional SQL. First, by hiding actual primary keys and foreign keys, there is no need to refer to the actual columns when performing table joins. Second, using SQL^{path} and "path expression" can speed up query design by avoiding explicitly written column constraints, especially for tables with complex keys. Particularly, joins spanning multiple tables can be replaced by a single compressed "path expression". This can simplify database queries and make them more readable and easier to understand for other developers, as well as reduce the likelihood of making mistakes thanks to shortened and simplified query syntax.

Naturally, there are two metrics to evaluate users' performance with a query language: readability and writability. Readability denotes the easiness of comprehending a query written by someone else in the specific query language, and can be measured by letting users translate given queries to English to explain their purposes. Furthermore, its performance can be quantified in two ways. If SQL^{path} is easier to use, it should not only take a user

less time, but also generate fewer errors when reading queries written in it. To verify it statistically, we present the following two null hypotheses to represent its time consumption and correctness measurements.

Null Hypothesis 1 (Readability$_{\text{NH}}$: Time Consumption)**.** *The average time taken to read and translate a query written in SQL^{path} to English is the same as when it is written in SQL.*

Null Hypothesis 2 (Readability$_{\text{NH}}$: Correctness)**.** *The average correctness mark earned for reading and translating a query written in SQL^{path} to English is the same as when it is written in SQL.*

The definition of time taken and correctness mark will be explained later in the study design section. Note that all queries we referred to previously and in the rest of this paper are conjunctive with the addition of the `not` operator. On the other hand, writability reflects how easily one can design and write a query with the given query language to solve a specific problem. Two similar null hypotheses are presented here to measure the time consumption and correctness perspectives of writability.

Null Hypothesis 3 (Writability$_{\text{NH}}$: Time Consumption)**.** *The average time taken using SQL^{path} to design and write a conjunctive query to solve a given problem is the same as using SQL.*

Null Hypothesis 4 (Writability$_{\text{NH}}$: Correctness)**.** *The average correctness mark rewarded for using SQL^{path} to design and write a conjunctive query to solve a given problem is the same as using SQL.*

These four null hypotheses capture the two aforementioned advantages respectively, and characterize them with quantitative attributes: time consumption and correctness, which can in turn be measured empirically. To verify their validity, we specifically designed the experiment to collect data, and performed statistical analysis to test whether none, some, or all of these hypotheses should be rejected. This can help us answer the original question of whether SQL$^{\text{path}}$ possesses its claimed benefits in real world applications. For example, if SQL$^{\text{path}}$ was more efficient and faster to read when compared with SQL, we should be able to reject Null Hypothesis 1 with an alternative hypothesis that is backed by data where the corresponding time consumption of SQL is statistically significantly greater than SQL$^{\text{path}}$.

4.2 Study Design

4.2.1 Study Population

For this study, we recruited participants that fit the targeted user population of experienced SQL users to have a more realistic and reliable evaluation of the abstract relational model and SQL$^{\text{path}}$. While we conducted the experiment at University of Waterloo, the most fitting study population of experienced SQL users were students who had taken database related courses that covered SQL queries before. Additionally, we preferred participants that were still using SQL regularly to avoid students who had put SQL aside for extended time in case their memory of SQL had become rustic and no longer represented our targeted user base. Moreover, we tried to recruit both undergraduate students and graduate students to evaluate the ability to adopt abstract relational model and SQL$^{\text{path}}$ for participants of different academic levels and background.

Inclusion Criteria

For undergraduate participants, we only included the ones who were taking CS348 - "Introduction to Database Management" at University of Waterloo at the experiment time. Additionally, we made sure the experiment took place right after they finished all SQL related materials, while their knowledge of SQL queries were still fresh in their minds. For their graduate counterparts, we only recruited from the data systems group (DSG), as their researches were related to database systems. More specifically, we asked and made sure they were using SQL queries regularly.

Recruitment Process

To recruit our sample participants, in-class recruitment sessions were held for undergraduate students in both sections of the CS348 class at University of Waterloo during the Spring 2017 term, right after SQL related materials were covered and associated assignments were done. Students were encouraged to join our study to become part of a real scientific experiment as well as obtain an extra opportunity to practice what they had learned in class to prepare for their midterm exams. Recruitment envelopes were distributed to students in class, where all the information regarding the experiment, for example, times, locations and how to register, were stated. We held two experiment sessions at different times but at the same location in order to make sure everyone that were interested could join the experiment.

For graduate students, in person recruitment was performed towards members of the DSG thanks to the small size of the targeted group. Based on time availability, they were formed into small groups of size up to 5 for different experiment sessions and they were notified of the experiment date, time, location by email. Additionally, all participants were rewarded a $15 gift card each after the experiment as an appreciation of their time. In the end, 9 undergraduate and 15 graduate students were recruited.

4.2.2 Instruments

We created two sets of questions to evaluate the performance of SQL^{path} in the readability and writability perspectives, respectively. Each set contained three questions, ordered by increasing difficulty, where level of difficulty was ranked by the complexity of constraints, and number of tables that were required to be joined to yield the correct solution. By setting up questions this way, we tried to first help the experiment participants become familiar with the problem domain and the concept of SQL^{path}, before taking on questions that were more complex and challenging.

Moreover, all participants worked on readability questions first before moving on to writability ones. We arranged the question sets in this particular order to make sure participants could first have access to some query examples to get familiar with the query language, in an attempt to minimize their likelihood to produce syntax mistakes. Since we were primarily interested in conjunctive queries, the core idea of all the questions were based on table joins and the understanding of primary keys, foreign keys and their usage in database queries.

Readability Questions

To evaluate whether complex queries are simpler to express and easier to understand with SQL^{path}, the first set of questions were composed of three pieces of query code, written with either SQL or SQL^{path}, where experiment participants were asked to explain what questions the queries were trying to answer by translating them into English. The three readability questions we used is shown in Figure 4.1, where questions for SQL are listed on the left side and questions for SQL^{path} are listed on the right side. We mostly focused on the basics of conjunctive queries (extended to include the `not` operator) and the most difficult part was nested sub-queries with the "exists" operator.

Q1. Describe what the following database query computes in English.

```
select distinct snum, sname
from student s
where exists (
      select * from enrollment e
      where e.snum = s.snum
      and e.deptcode = 'CS'
      and (e.cnum = 341 or e.cnum = 348))
```

Q2. Describe what the following database query computes in English. [**hour**(time) returns the hour part of a given time as an integer between 0 to 23]

```
select distinct pnum, pname
from professor p
where exists (
      select * from class cl
      where cl.deptcode = 'CS'
      and cl.cnum = 348
      and cl.pnum = p.pnum)
and not exists (
      select * from class cl, schedule sch
      where cl.pnum = p.pnum
      and cl.deptcode = sch.deptcode
      and cl.cnum = sch.cnum
      and cl.term = sch.term
      and cl.section = sch.section
      and hour(sch.time) < 12
)
```

Q3. Describe what the following database query computes in English.

```
select distinct p.pnum, p.pname
from professor p
where exists (
     select * from department d
     where d.deptcode = p.deptcode
     and exists (
           select * from courses c
           where c.deptcode = d.deptcode
           and exists (
           select * from mark m
           where m.grade = 100
           and m.cnum = c.cnum
           and m.deptcode = c.deptcode
           )
     )
)
```

Q1. Describe what the following database query computes in English.

```
select distinct e.student.snum, e.student.sname
from enrollment e
where e.class.course.department.deptcode = 'CS'
and (e.class.course.cnum = 341 or e.class.course.cnum =
348)
```

Q2. Describe what the following database query computes in English. [**hour**(time) returns the hour part of a given time as an integer between 0 to 23]

```
select distinct cl.professor.pnum, cl.professor.pname
from class cl
where cl.course.cnum = 348
and cl.course.department.deptcode = 'CS'
and not exists (
  select * from schedule sch
  where sch.class.professor = cl.professor
  and hour(sch.time) < 12
)
```

Q3. Describe what the following database query computes in English.

```
select distinct p.pnum, p.pname
from professor p
where exists (
    select * from mark m
    where m.grade = 100
    and m.enrollment.class.course.department =
  p.department
)
```

Figure 4.1: Readability Questions: SQL(left) and $\mathrm{SQL}^{\mathrm{path}}$(right)

Q4. Write a SQL query that implements the following:

Find the distinct number (pnum) and name (pname) of all the professors from the department named 'Computer Science' that have taught a class offered by a different department (with different deptcode).

Q5. Write a SQL query that implements the following:

Find the distinct numbers (snum) and names (sname) of all students who have taken Grant Weddell's CS348 and received the mark greater than 90.

Q6. Write a SQL query that implements the following:

Find the distinct deptcode, cnum, and cname of all the courses that have failed (received grade less than 50) some students in its classes offered in the morning (the class has at least one schedule with **hour(**time**)** < 12) that are currently in their second year and have enrolled in a class taught by 'David Toman' (either previously or currently).

Figure 4.2: Writability Questions: SQL

Writability Questions

For the second set of questions, participants were asked to design and write down database queries either using SQL or SQL$^{\text{path}}$, to solve specific problems looking for attributes of entities satisfying certain constraints. All problems were based on the given relational database model or abstract relational model shown previously. For each individual question regarding SQL and SQL$^{\text{path}}$, the problem explanation was identical, while the only difference was how the expected solution is going to be expressed. The query language itself would not affect the underlying logic of the given question. The set of writability questions for SQL are shown in Figure 4.2. Questions for SQL$^{\text{path}}$ is skipped here, as the only difference was that the usage of attribute paths were allowed to write such queries.

4.2.3 Measurement

Response Variables

Based on our null hypotheses, we are interested in two performance metrics: time consumption and correctness. These metrics can reflect whether SQL$^{\text{path}}$ is faster and more accurate to read and write when working with conjunctive and alike database queries, and

are considered as response variables to our experiment. During the experiment, a participant would work with either SQL or SQL^{path} at a given time, and was not allowed to go back to a previously question once it was finished. Therefore, their time consumption and correctness marks for each question, with a specific query language can be easily recorded and evaluated.

Time consumption can be measured for each individual question by calculating the time difference in seconds between its starting time and ending time, which was logged by each participant during the experiment. If SQL^{path} was faster and easier to understand, we would expect participants to consume less time on the set of readability questions expressed in SQL^{path}. The same can be said about the set of writability questions, which were used in the experiment to evaluate the usability of SQL^{path} in query design.

On the other hand, correctness is marked by three human markers in accordance to a predefined marking scheme. We use a finer-grain marking scheme, as we are interested in measuring the scale of how much more or less accurate SQL^{path} solutions were when compared with their SQL counterparts. We ask the three markers to go through all participants' solutions and evaluate their correctness independently on a scale from 0 to 4, following the guideline shown below:

- 0 - the solution is completely wrong;
- 1 - the solution does not solve the question at all, but it can be observed that the participant has grasped the basic concept of SQL or SQL^{path};
- 2 - the solution contains mistakes, but is on right track and joined most of the required tables correctly;
- 3 - the solution is mostly correct, and it may only contain minor mistakes;
- 4 - the solution solves the question completely and correctly.

Usage of half points are allowed if the marker deem the correctness of a solution fits between two levels. In the end, the averaged marks is used as the final evaluation of correctness. For readability questions, the average correctness marks are used as the sole measurement of the correctness of solutions.

For writability questions, in addition to the previously defined scalar correctness grade points, we have two extra labels for each question, recording participants' syntax errors and inconvenience. A solution is deemed to contain syntax error if any of the three markers can find syntax mistakes in it. On the other hand, a solution is considered to be inconvenient

if any of the three markers can detect places where the participants used special notations such as ellipsis or explanatory words to shorten their solutions, hence their solutions are not syntactically correct. The combination of correctness grades and detection of syntactical mistakes and inconvenience is used to reflect the correctness aspect of query writing questions. Moreover, to make the marking process as fair as possible, the order of the query language to mark for each marker is alternated. For example, if marker one marks SQL first, marker two will mark SQL^{path} first.

Explanatory Variables

The first and most important explanatory variable for each question is the query language the given participant is asked to work with. This information was collected at the question designing phase, as the query language itself was specified as part of the questions. It is supposed to be the major factor affecting participants' time consumption and correctness of their solutions. For both readability and writability questions, if the proposed null hypotheses held, then the usage of SQL^{path} should not change participants' time consumption and correctness much.

Additionally, information regarding participants' native speaking language was also collected as an explanatory variable to the experiment, as it might affect the learning speed of SQL^{path} and the participants ability to understand the questions. Among the 9 undergraduate participants, two of them were native English speakers, while 5 of the 15 graduate participants were native English speakers. The participants' education level was also an explanatory variable.

We also have an extra explanatory variable that measures participants' prior experience with SQL. For undergraduate students, we asked them whether they had previous experience working with RDBMS and SQL before taking the CS348 class, such as internships. We suspect that prior experience may affect their ability and speed to learn abstract relational model and SQL^{path}. For graduate students, we specifically asked them whether they have experience working with RDBMS and SQL in their study before. We suspect that if some of the graduate students' research directions had no relationship to SQL, they might not be as familiar with database queries as their peers. In the collected results, three of the undergraduate participants had previous experience with SQL while all of our recruited graduate students had previous experience with SQL.

4.2.4 Experimental Design

While we treated undergraduate and graduate students slightly differently in their respective session, the details of their separate experiment process is explained below. The entire experiment was conducted anonymously where participants' names were never collected nor stored.

Undergraduate Students

We held two sessions for undergraduate students. They both happened after students finishing the first assignment which was related to SQL programming and before their midterm exam. We had 4 students participated in session one and 5 students participated in session two.

For each session, undergraduate students were invited to the experiment and were asked to fill in a questionnaire. In the questionnaire, they were assigned a unique study id and asked to answer two very simple questions regarding their native speaking language and experience working with SQL before taking the database course. Once finished the questionnaire, students were asked to keep one copy of their study id and the experimenter collected all the questionnaires. After that, the experimenter divided participants into four sets: students with previous experience and whose first language was English, students with previous experience and whose first language was not English, students without previous experience and whose first language was English, and students without previous experience and whose first language was not English. Next, we randomly shuffled the study ids in each of the four set of participants into two partitions. The students whose study id fell into the first partition were assigned to Group A while the rest were assigned to Group B. During this process, we tried to balance the overall number of students in these two groups. For example, if the number of students in one set was odd, we would first assign more to Group A. If we later encountered another set with odd number of students, we would try to balance it by assigning more participants from this set to Group B.

Students in Group A were asked to solve their given questions with SQL while the ones in Group B were asked to solve a similar set of questions with SQL^{path}. Combining two sessions, we had 4 participants worked with SQL and 5 participants worked with SQL^{path}. Each student in Group A was given an envelope containing the experiment instruction, technical background and instruction of SQL and relational model, a specific relational model illustrating the problem domain, and a set of questions. Similarly, students in Group B obtained an envelope with similar experiment instruction, technical background

and instruction of SQL^{path} and an abstract relational model illustrating the same problem domain, and a similar set of questions.

When the experimenter distributed the envelope based on study id, the answers to the two questions in participants' corresponding questionnaires as well as the study id were recorded on the envelope. Participants first spent 5 minutes reading the experiment instruction, and then given ten minutes to read their own technical instructions that demonstrate how SQL/SQL^{path} and relational/abstract relational model works. In the mean while, they would go through some examples explaining the proper way to solve both types of questions. The corresponding relational/abstract relational diagram that illustrated the problem domain (entities with attributes and relations that were needed to solve the questions) would later be used in the problem solving phase. After the experimenter started the problem solving phase, each participant was given an hour to solve all six questions, in the sequential order provided (order number was labeled on the questions beforehand). Readability questions were always solved before writability questions, and once a question is finished, the participant was not allowed to return to it. The start and end time for solving each question was logged by the student in the format of "hour:minute:second". The participants were asked to place everything back into the envelope and return it to the experimenter after the experiment. Their solutions were then collected and stored for marking later.

Graduate Students

Different from undergraduate students, although the questions were still the same, graduate students were asked to work in small sized groups with the experimenter (at most 5 people in one session). Furthermore, each graduate participant was asked to solve the questions with both SQL and SQL^{path}. Therefore, their experiment session was divided into two parts where the participants either worked on SQL questions or SQL^{path} questions.

We had 15 graduate participants in total. Similar to the experiment among undergraduate students, graduate students were asked to fill in the questionnaire with a unique study id and the same two questions regarding their first language and prior experience with SQL. After that, participants kept their own study id and the experimenter collected all the questionnaires. The experimenter then shuffled the study ids and students were assigned by their ids alternatively into Group A and Group B. For example, the first id on the pile after shuffling was assigned into Group A and the second was assigned into Group B, so on so forth. Afterwards, Group A (total of 8) worked on SQL first while the other 7 (Group B) worked on SQL^{path} first.

Each graduate participant received two envelopes, A and B. For participants in Group A, envelope A contained the same material as the undergraduate participants who worked on SQL were given, and envelope B contained the same material as the undergraduate participants who worked on SQL^{path} were given. The materials given for participants in Group B was in the opposite order. The experimenter then wrote down the study id on both envelopes and recorded the answers to the questions on the questionnaire, too. After reading the experiment instruction, different from the undergraduate experiment sessions, the experimenter spent 10 mins to explain an example with both SQL and SQL^{path} solutions to give a general idea of these two query languages. Then, participants were allowed to open their own envelope and read the technical instruction inside. Next, the participants were asked to start solving the questions sequentially in the same order using the given relational/abstract relational diagram as their undergraduate counterparts. Once they finished envelope A, the first part of experiment was done. Participants could take a short break and then start the second part. During the second part of the experiment, envelope B was opened and the process was the same as when they were working with envelope A. Same to undergraduate experiment, the start and end time for solving each question was logged by the student in the format of "hour:minute:second". By having two groups that started with different query languages, we hope to eliminate the bias introduced by familiarity when working on second set of questions. After they were done, everything was put into envelopes and the solutions were still collected and organized anonymously.

The relational/abstract relational model and database schemas used by both undergraduate and graduate participants to solve SQL and SQL^{path} questions were shown previously in Figure 2.1 and Figure 3.1, while the readability and writability questions are shown in Figure 4.1 and Figure 4.2, respectively.

4.3 Data

After all participants' solutions were collected and their correctness was graded by the three markers, the final results were aggregated and organized for further statistical analysis. The raw data for undergraduate participants is displayed in Table 4.1 for readability questions and Table 4.3 for writability questions, while Table 4.2 and Table 4.4 hold the raw data for graduate participants. We separate data from graduate students and undergraduate students here due to the different setups of their respective experiment sessions.

In all four tables, the "ID" column specifies a participant's assigned study ID in the experiment; "English" column denotes whether a participant is a native English speaker; and "Query Type" column indicates either a participant was working with SQL or SQL^{path}

to solve the given questions. For the "Experience" column, it is "True" for an undergraduate student if he had previous experience working with SQL prior to taking class CS348. For graduate students, since they all had experience working with SQL before in their research, this column is discarded. However, graduate participants has a special "SQL First" column indicating whether one worked with SQL first or SQL$^{\text{path}}$ first at his own experiment session.

Additionally, the "Start" and "End" columns for each question are used to record the starting and ending time of the question logged by the student during the experiment. The values are in the format of "hour:minute:second". The following "Diff" column contains the associated time difference between the starting and ending time, in seconds. It will later be used as the measurement of time consumption in our analysis. The "Grade" column for each question lists the raw grade marks given by all three markers, without averaging. For writability questions, we also include two extra columns of "Syntax" and "Inconvenience", indicating the discovery of syntax errors and purposely shortened solutions, independently from the three markers, in the same order as the "Grade" column. These tables display the raw information we collected from the experiment, without any further processing. All of our analysis in the following section is based on the results contained in them.

ID	English	Experience	Query Type	Readability: Q1				Readability: Q2				Readability: Q3			
				Start	End	Diff	Grade	Start	End	Diff	Grade	Start	End	Diff	Grade
1	True	True	SQL$^{\text{path}}$	17:09:44	17:11:27	103	3/4/4	17:11:37	17:14:00	143	3/4/4	17:14:21	17:15:34	73	4/4/4
2	False	False	SQL	17:07:48	17:10:24	156	4/4/4	17:10:43	17:15:11	268	2/3/2	17:15:26	17:19:33	247	1/4/2
3	False	True	SQL	17:15:20	17:17:14	114	4/4/4	17:18:18	17:22:01	223	2/3/2	17:23:30	17:28:03	273	0/2/2
4	False	True	SQL$^{\text{path}}$	17:09:18	17:11:21	123	4/3/3	17:12:07	17:14:26	139	2/3/2	17:14:40	17:15:58	78	2/2/2
5	False	False	SQL	17:14:23	17:15:59	96	4/4/4	17:16:27	17:18:51	144	1/3/2	17:19:14	17:21:45	151	0/2/2
6	False	False	SQL$^{\text{path}}$	17:13:48	17:15:12	84	4/4/4	17:17:06	17:20:20	194	1/2/2	17:20:48	17:29:06	498	4/4/4
7	True	False	SQL$^{\text{path}}$	17:15:22	17:18:17	175	3/4/3	17:18:37	17:22:59	262	1/2/2	17:23:19	17:25:40	141	3/4/4
8	False	False	SQL$^{\text{path}}$	17:14:47	17:18:05	198	4/4/4	17:18:22	17:21:19	177	2/2/2	17:21:34	17:24:17	163	4/4/4
9	False	False	SQL	17:15:43	17:17:11	88	3.5/4/4	17:17:36	17:19:13	97	1.5/2/2	17:19:35	17:21:53	138	0/2/2

Table 4.1: Experiment Result: Readability Questions, Undergraduate Students

4.4 Analysis

Based on our null hypotheses, we expect the usage of SQL and SQL$^{\text{path}}$ to be the major deciding factor of a participant's efficiency, which can be measured by time consumption, and accuracy, which can be measured by correctness marks. However, one may suspect that there is some internal correlation between time consumption and correctness themselves. For example, a participant may achieve a higher mark on a question due to spending more time to think through the question more thoroughly. If that was the case, we would not be

ID	English	SQL First	Query Type	Readability: Q1				Readability: Q2				Readability: Q3			
				Start	End	Diff	Grade	Start	End	Diff	Grade	Start	End	Diff	Grade
10	True	True	SQL	13:11:23	13:14:16	173	4/4/4	13:14:53	13:19:40	287	3/3/4	13:19:57	13:26:55	418	4/4/4
10	True	True	SQL^{path}	14:02:29	14:03:33	64	3.5/4/3	14:03:55	14:06:00	125	1/3/3	14:06:20	14:07:48	88	4/4/4
11	False	False	SQL^{path}	13:43:27	13:45:37	130	4/4/4	13:46:27	13:50:05	218	2/3/3	13:52:38	13:55:21	163	3/4/4
11	False	False	SQL	14:29:29	14:30:30	61	4/4/4	14:30:41	14:34:55	254	2/3/4	14:35:10	14:36:59	109	4/4/4
12	False	True	SQL	13:42:53	13:45:13	140	4/4/4	13:45:26	13:49:23	237	1/3/2	13:49:36	13:54:11	275	0/2/2
12	False	True	SQL^{path}	14:33:05	14:35:06	121	4/4/4	14:35:21	14:38:24	183	1/4/4	14:39:37	14:41:55	138	0/2/2
13	False	False	SQL^{path}	13:43:00	13:46:00	180	3/2/4	13:47:50	13:49:20	90	4/4/4	13:49:40	13:51:59	139	4/4/4
13	False	False	SQL	14:14:15	14:16:45	150	4/4/4	14:17:10	14:20:10	180	4/4/4	14:20:20	14:23:15	175	4/4/4
14	False	True	SQL	13:11:07	13:12:50	103	3/4/4	13:13:17	13:16:30	193	2/3/2	13:17:37	13:20:36	179	0/2/3
14	False	True	SQL^{path}	13:43:17	13:45:25	128	3/4/4	13:45:45	13:47:20	95	3/4/4	13:47:36	13:49:26	110	4/4/4
15	False	False	SQL^{path}	13:12:50	13:13:50	60	4/4/4	13:14:10	13:15:05	55	3/4/4	13:15:15	13:15:50	35	0/1/1
15	False	False	SQL	13:24:40	13:25:21	41	4/4/4	13:25:32	13:26:28	56	4/4/4	13:26:40	13:27:50	70	0/2/2
16	False	True	SQL	13:09:55	13:13:40	225	2/4/4	13:14:06	13:18:02	236	4/3/4	13:19:24	13:23:38	254	0/2/1
16	False	True	SQL^{path}	13:55:20	13:58:50	210	4/4/4	14:06:50	14:09:38	168	4/4/4	14:10:16	14:12:40	144	0/3/2
17	True	False	SQL^{path}	13:07:10	13:08:03	53	3/4/4	13:08:26	13:11:15	169	1/2/2	13:11:57	13:13:00	63	0/2/2
17	True	False	SQL	13:49:25	13:50:12	47	4/4/4	13:50:24	13:52:37	133	4/4/4	13:52:53	13:55:06	133	4/4/4
18	True	True	SQL	13:06:28	13:08:17	109	3.5/4/4	13:08:33	13:10:23	110	1/3/2	13:10:46	13:13:01	135	0/2/2
18	True	True	SQL^{path}	13:50:49	13:51:24	35	2.5/4/3	13:51:45	13:53:14	89	3.5/4/4	13:53:40	13:54:38	58	3.5/4/4
19	False	False	SQL^{path}	13:18:50	13:20:20	90	3/4/3	13:21:22	13:22:30	68	1/2/2	13:22:49	13:24:28	99	1/4/4
19	False	False	SQL	13:48:21	13:49:07	46	3/3/4	13:49:42	13:51:40	118	1/2/1	13:52:00	13:53:09	69	0/2/2
20	False	True	SQL	13:32:05	13:34:14	129	4/4/4	13:34:32	13:38:04	212	2/3/2	13:16:06	13:19:11	185	4/4/4
20	False	True	SQL^{path}	13:48:27	13:49:41	74	4/4/4	13:50:01	13:51:14	73	1/2/2	13:51:34	13:53:42	128	3/4/4
21	False	False	SQL^{path}	13:17:17	13:19:59	162	3/4/4	13:20:10	13:22:43	153	1/2/2	13:23:04	13:25:05	121	4/4/4
21	False	False	SQL	13:47:12	13:48:08	56	3/3/4	13:48:30	13:50:24	114	2/3/2	13:50:50	13:52:05	75	4/4/4
22	True	True	SQL	13:21:30	13:23:32	122	4/4/4	13:24:22	13:30:00	338	3/3/3	13:32:41	13:35:10	149	0/2/2
22	True	True	SQL^{path}	14:02:34	14:03:30	56	4/4/4	14:04:01	14:05:18	77	3/4/4	14:05:30	14:07:03	93	0/2/2
23	False	False	SQL^{path}	13:14:27	13:16:26	119	2.5/3/3	13:16:50	13:19:13	143	2/2/2	13:19:25	13:20:55	90	0/2/2
23	False	False	SQL	13:40:12	13:41:57	105	3/3/4	13:42:09	13:45:56	227	4/4/4	13:46:08	13:51:45	337	1/3/1
24	True	True	SQL	13:05:07	13:05:40	33	2/3/3	13:05:48	13:06:45	57	1/2/2	13:06:51	13:07:42	51	0/2/2
24	True	True	SQL^{path}	13:32:03	13:32:28	25	2/3/2	13:32:35	13:33:25	50	2/4/4	13:33:44	13:35:07	83	2.5/4/4

Table 4.2: Experiment Result: Readability Questions, Graduate Students

able to explain the correctness difference from the selection of query languages. Therefore, before diving any further, we want to first test whether such a correlation can be validated statistically. In all of our following analysis, the correctness marks from three markers are averaged to create a numerical measure.

4.4.1 Time Consumption vs. Correctness

We use scatterplots to draw all of the participants' time consumption and correctness marks, for each individual question. Furthermore, we split the graphs by query languages, in case dealing with the unfamiliar SQL^{path} may affect a participant's performance. The results for readability questions are shown in Figure 4.3 (SQL) and Figure 4.4 (SQL^{path}), while the results for writability questions are shown in Figure 4.5 (SQL) and Figure 4.6 (SQL^{path}). In the scatterplots, we use x-axis to represent time consumption in seconds, and y-axis to represent the averaged correctness marks, as we defined previously. A dot in a graph represents one participant's time consumption and correctness mark for one question,

ID	English	Experience	Query Type	Writability: Q4						Writability: Q5						Writability: Q6					
				Start	End	Diff	Grade	Syntax	Inconvenience	Start	End	Diff	Grade	Syntax	Inconvenience	Start	End	Diff	Grade	Syntax	Inconvenience
1	True	True	SQL^{path}	17:15:49	17:21:04	315	3/3/2	0/1/1	0/0/0	17:21:20	17:28:42	442	4/3/3	0/1/0	0/0/0	17:28:59	17:45:41	1002	3/3/2	0/1/0	0/0/0
2	False	False	SQL	17:19:46	17:23:23	217	3/3/3	1/0/0	0/0/0	17:23:39	17:30:04	385	4/4/4	0/0/0	0/0/0	17:30:16	17:40:58	642	4/3/4	0/0/0	0/0/0
3	False	True	SQL	17:28:27	17:32:30	243	3/3/3	0/0/0	0/0/0	17:33:14	17:40:41	447	4/4/4	0/0/0	0/0/0	17:41:02	17:51:03	601	4/3/4	0/0/0	0/0/0
4	False	True	SQL^{path}	17:16:17	17:19:20	183	0/2/2	0/0/0	0/0/0	17:19:34	17:23:24	230	2/3/2	1/1/0	0/0/0	17:23:48	17:33:19	571	1/3/2	1/1/1	0/0/0
5	False	False	SQL	17:22:11	17:26:54	283	3.5/4/4	1/0/0	0/0/0	17:27:17	17:32:49	332	3.5/4/4	0/0/0	0/0/0	17:33:21	17:49:51	990	4/4/4	0/0/0	0/0/0
6	False	False	SQL^{path}	17:29:32	17:36:58	446	4/4/4	0/0/0	0/0/0	17:37:45	17:43:06	321	4/4/4	0/0/0	0/0/0	17:44:20	17:54:14	594	2/3/4	0/0/0	0/0/0
7	True	False	SQL^{path}	17:25:59	17:32:42	403	4/4/4	1/0/0	0/0/0	17:33:06	17:38:17	311	4/4/4	1/0/0	0/0/0	17:38:40	17:49:57	677	2/4/3	0/1/0	0/0/0
8	False	False	SQL^{path}	17:24:35	17:25:26	51	3/4/4	1/0/0	0/0/0	17:28:41	17:32:32	231	3/3/4	0/0/0	0/0/0	17:32:47	17:40:59	492	2/4/3	0/0/0	0/0/0
9	False	False	SQL	17:22:08	17:28:06	358	1/4/3	1/0/0	0/0/0	17:28:25	17:31:44	199	2/4/4	1/0/1	0/0/0	17:32:05	17:42:22	617	1/4/4	0/0/0	0/0/0

Table 4.3: Experiment Result: Writability Questions, Undergraduate Students

ID	English	SQL First	Query Type	Writability: Q4						Writability: Q5						Writability: Q6					
				Start	End	Diff	Grade	Syntax	Inconvenience	Start	End	Diff	Grade	Syntax	Inconvenience	Start	End	Diff	Grade	Syntax	Inconvenience
10	True	True	SQL	13:27:42	13:31:35	233	4/4/4	0/0/0	0/0/0	13:31:59	13:46:39	880	4/4/4	0/0/0	0/0/0	13:47:39	14:01:10	811	3/4/4	0/0/0	0/0/0
10	True	True	SQL^{path}	14:08:20	14:12:28	248	4/4/4	0/0/0	0/0/0	14:12:53	14:20:44	471	4/3/4	0/0/0	0/0/0	14:20:54	14:28:37	463	3/4/4	0/1/0	0/0/0
11	False	False	SQL^{path}	13:55:42	14:03:15	453	3/4/3	0/0/0	0/0/0	14:03:30	14:09:47	377	4/4/4	0/0/0	0/0/0	14:10:05	14:28:10	1085	4/4/4	0/0/1	0/0/0
11	False	False	SQL	14:37:34	14:47:16	582	2/3/4	0/0/0	0/0/0	14:47:28	14:53:16	348	4/4/4	0/0/0	0/0/0	14:53:30	15:06:20	770	0/4/3	0/0/0	0/0/0
12	False	True	SQL	13:54:25	14:04:27	602	1/2/1	0/0/0	0/0/0	14:04:41	14:12:08	447	4/4/4	0/0/0	0/0/0	14:12:24	14:32:09	1185	4/4/4	0/0/0	0/0/0
12	False	True	SQL^{path}	14:42:11	14:46:11	240	2/3/3	0/0/0	0/0/0	14:46:27	14:53:09	402	3.5/4/4	0/0/0	0/0/0	14:53:22	15:00:00	398	3/4/4	0/0/0	0/0/0
13	False	False	SQL^{path}	13:52:15	13:55:30	195	2/3/4	0/0/0	0/0/0	13:55:40	14:00:10	270	3/4/4	1/0/0	0/0/0	14:00:30	14:12:45	735	2/3/4	1/0/0	0/0/0
13	False	False	SQL	14:23:30	14:26:59	209	4/4/4	1/0/0	0/0/0	14:27:25	14:35:50	505	2/4/4	0/0/0	0/0/0	14:36:10	14:50:35	865	4/4/4	0/0/0	0/0/0
14	False	True	SQL	13:21:38	13:26:20	282	3/4/4	1/0/0	0/0/0	13:26:51	13:32:16	325	2/3/3	0/1/1	1/0/0	13:32:34	13:42:09	575	2/2/2	0/0/1	1/0/0
14	False	True	SQL^{path}	13:49:43	13:53:16	213	2/4/2	0/0/0	0/0/0	13:53:20	13:57:32	252	4/3/4	0/1/1	1/0/0	13:57:45	14:02:43	298	1/2/3	1/1/1	0/0/0
15	False	False	SQL^{path}	13:16:00	13:18:07	127	2/3/3	1/1/1	0/0/0	13:18:20	13:20:00	100	2/2/1	1/1/1	0/0/0	13:20:11	13:23:18	187	1/2/3	1/0/1	0/0/0
15	False	False	SQL	13:28:05	13:29:45	100	1/3/2	1/0/0	0/0/0	13:30:00	13:31:50	110	2/2/1	0/0/0	0/0/0	13:33:53	13:38:25	272	1/3/1	1/1/1	0/0/0
16	False	True	SQL	13:24:01	13:32:48	527	3/4/3	0/0/0	0/0/0	13:33:07	13:40:05	418	2/3/2	0/0/0	0/0/0	13:40:26	13:54:26	840	2/4/1	1/0/1	1/0/0
16	False	True	SQL^{path}	14:13:10	14:19:10	360	0/2/1	0/1/0	0/0/0	14:23:00	14:25:05	125	3/2/1	1/1/1	0/0/0	14:25:26	14:28:48	202	0/2/2	1/1/1	0/0/0
17	True	False	SQL^{path}	13:13:01	13:18:58	357	3.5/4/4	0/0/0	0/0/0	13:19:11	13:24:30	319	4/4/4	0/0/0	0/0/0	13:24:52	13:47:43	1371	4/4/3	0/0/0	0/0/0
17	True	False	SQL	13:55:20	13:59:13	233	4/4/4	0/0/0	0/0/0	13:59:41	14:09:19	578	3/4/4	0/0/0	0/0/0	14:09:35	14:16:30	415	4/4/4	0/0/0	0/0/0
18	True	True	SQL	13:13:35	13:17:18	223	4/4/4	0/0/0	0/0/0	13:17:56	13:29:43	707	2.5/3/4	0/1/0	0/0/0	13:30:07	13:49:25	1158	3/3/1	1/1/1	0/0/0
18	True	True	SQL^{path}	13:54:59	13:57:40	161	3/4/4	1/0/0	0/0/0	13:58:02	14:01:18	196	4/4/4	0/0/0	0/0/0	14:01:33	14:06:49	316	3.5/4/4	1/0/0	0/0/0
19	False	False	SQL^{path}	13:25:15	13:29:25	250	4/4/4	0/0/0	0/0/0	13:29:36	13:34:58	322	4/4/4	0/0/0	0/0/0	13:35:23	13:45:28	605	2/4/4	1/0/0	0/0/0
19	False	False	SQL	13:54:30	13:57:30	180	2/3/4	0/0/0	0/0/0	13:57:00	14:00:08	188	0/2/1	0/0/0	0/0/0	14:01:05	14:07:55	410	2/3/2	0/1/1	1/1/1
20	False	True	SQL	13:19:27	13:24:35	308	4/4/4	1/0/0	0/0/0	13:25:01	13:31:43	402	1/3/1	0/1/0	0/0/0	13:38:19	13:45:15	416	1/4/3	0/0/0	0/0/0
20	False	True	SQL^{path}	13:53:57	13:57:59	242	4/4/4	0/0/0	0/0/0	13:58:17	14:02:21	244	4/4/4	0/0/0	0/0/0	14:02:38	14:11:01	503	1/3/3	0/0/0	0/0/0
21	False	False	SQL^{path}	13:25:32	13:28:52	200	4/4/4	0/0/0	0/0/0	13:29:20	13:32:03	163	4/4/4	0/1/0	0/0/0	13:32:22	13:45:20	778	0/4/4	1/0/0	0/0/0
21	False	False	SQL	13:52:35	13:56:41	246	3.5/4/4	0/0/0	0/0/0	13:56:59	14:00:56	237	4/4/4	0/0/0	0/0/0	14:01:17	14:10:57	580	2/4/3	0/0/0	0/0/0
22	True	True	SQL	13:36:00	13:43:00	420	2/4/3	0/0/1	0/0/0	13:44:00	13:50:29	389	2/3/3	0/0/0	0/0/0	13:50:40	14:01:28	648	2/4/3	0/0/1	0/0/0
22	True	True	SQL^{path}	14:07:35	14:14:14	399	4/4/3	1/0/0	0/0/0	14:14:37	14:18:48	251	2/4/4	1/0/0	0/0/0	14:19:15	14:25:26	371	1/4/4	0/1/0	0/0/0
23	False	False	SQL^{path}	13:21:07	13:25:05	238	4/4/4	0/0/0	0/0/0	13:25:17	13:30:13	296	4/4/4	0/0/0	0/0/0	13:30:22	13:39:25	543	2/3/2	1/0/0	1/0/0
23	False	False	SQL	13:51:56	13:55:13	197	4/4/4	0/0/0	0/0/0	13:55:24	14:01:03	339	3/4/4	1/0/0	0/0/0	14:01:13	14:16:08	895	1/4/4	0/0/0	0/0/0
24	True	True	SQL	13:07:49	13:11:45	236	1/3/3	1/0/0	0/0/0	13:11:55	13:16:22	267	3/4/2	0/0/0	0/0/0	13:16:30	13:30:20	830	2/3/4	0/0/0	0/0/0
24	True	True	SQL^{path}	13:35:14	13:37:55	161	3.5/4/4	1/1/0	0/0/0	13:38:07	13:43:05	298	4/4/4	1/1/0	0/0/0	13:43:16	13:51:58	522	1/4/3	1/0/0	0/0/0

Table 4.4: Experiment Result: Writability Questions, Graduate Students

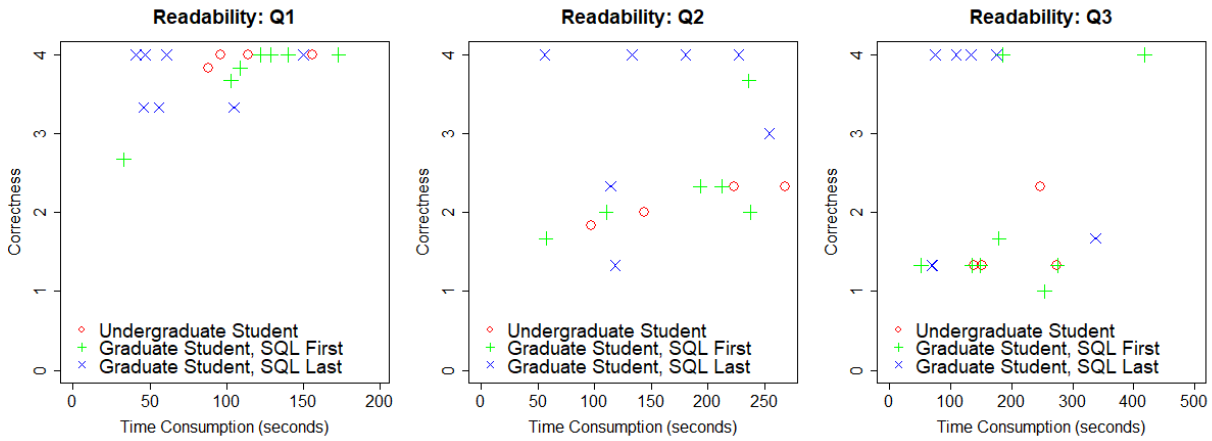

Figure 4.3: Readability Questions: Time Consumption vs. Correctness, with SQL

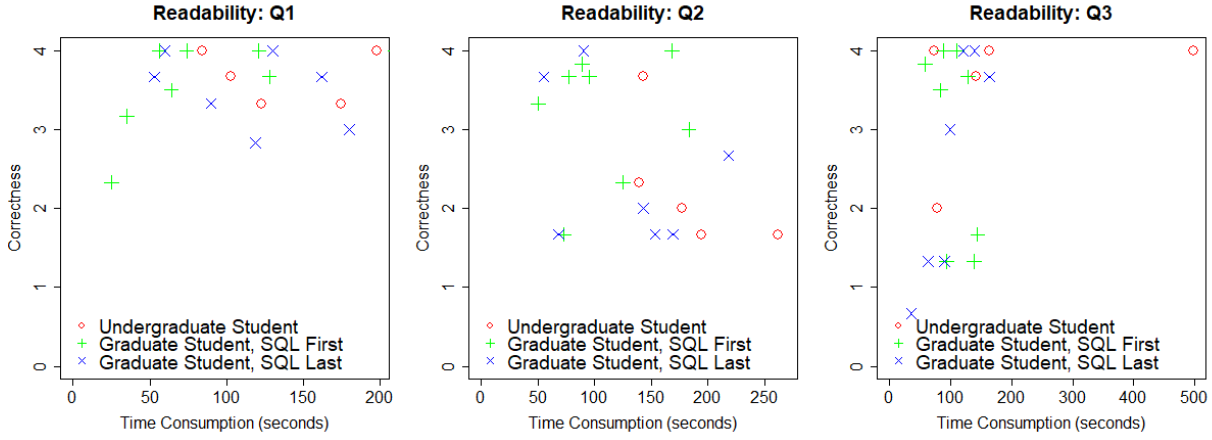

Figure 4.4: Readability Questions: Time Consumption vs. Correctness, with SQL^{path}

with either SQL or SQL^{path} as the used query language. Moreover, we use different color and dot type to denote undergraduate student, graduate student who worked with SQL first, and graduate student who worked with SQL^{path} first.

From these graphs, we are not able to see any clear relationship between time consumption and correctness. To verify it statistically, we aggregated all data points from both graduate (disregard the working order of SQL and SQL^{path}) and undergraduate stu-

dents in each graph and performed linear regression analysis upon them. For readability questions, the obtained p-values for SQL queries are Q1: 0.2816, Q2: 0.3207, Q3: 0.8073, and Q1: 0.3824, Q2: 0.0757, Q3: 0.1708 for SQL^{path} queries. For writability questions, the obtained p-values for SQL queries are Q4: 0.1506, Q5: 0.0896, Q6: 0.1507, and Q4: 0.7901, Q5: 0.0168, Q6: 0.0439 for SQL^{path} queries.

Other than the fifth and sixth questions of SQL^{path}, all other p-values are greater than 0.05 - the commonly used cutoff threshold. Hence overall it is statistically insignificant to suggest any linear relationship between time consumption and correctness. In another word, in most cases, spending more time on any of the questions will not produce more accurate answers nor the other way around, regardless the query language used. However, participants tend to write complex SQL^{path} queries better when they spend more time on them. Since the concept of SQL^{path} is new to them, spending more time may help them understand it better.

The only outlier is the fifth question of SQL^{path}, where its p-value is calculated to be 0.02017 and is below significance cutoff of 0.05. This suggests a positive linear relationship between time consumption and correctness for this specific question. In fact, its counterpart in SQL also has the lowest p-value among all questions. We suspect that the underlying logic of this type of question is rather straightforward. It may have multiple conditional constraints to satisfy, but contains less nesting (not as complicated as question six). Therefore, the reduction of correctness marks come more from missing constraints, which is less likely to happen when a participant spends more time double checking his answer. However, this still needs some formal verification. But as it was not the focus of this paper, we will not discuss it any further. Overall, the statistical results discovered here suggest no significant correlation between time consumption and correctness, and we can therefore attribute their changes to other explanatory variables, such as the query language used.

Additionally, the average time taken in seconds across all participants for SQL are 104.95 (Q1), 183.37 (Q2), 180.16 (Q3) seconds for readability questions and 298.89 (Q4), 394.89 (Q5), 711.58 (Q6) for writability questions. The average correctness marks are 3.75 (Q1), 2.71 (Q2), 2.24 (Q3) for readability questions and 3.26 (Q4), 3.16 (Q5), 3.04 (Q6) for writability questions. This confirms our design of questions that for both set of questions the difficulty level is sequentially increasing. However, for SQL^{path}, the average time taken in seconds are 109.50 (Q1), 133.55 (Q2), 125.25 (Q3) for readability questions, and 262.10 (Q4), 281.05 (Q5), 585.65 (Q6) for writability questions. The average correctness marks are 3.58 (Q1), 2.71 (Q2), 2.95 (Q3) for readability questions, and 3.30 (Q4), 3.54 (Q5), 2.86 (Q6) for writability questions. Although the order of measured difficulty is the same for writability questions, a more difficult readability question in SQL may not be as difficult

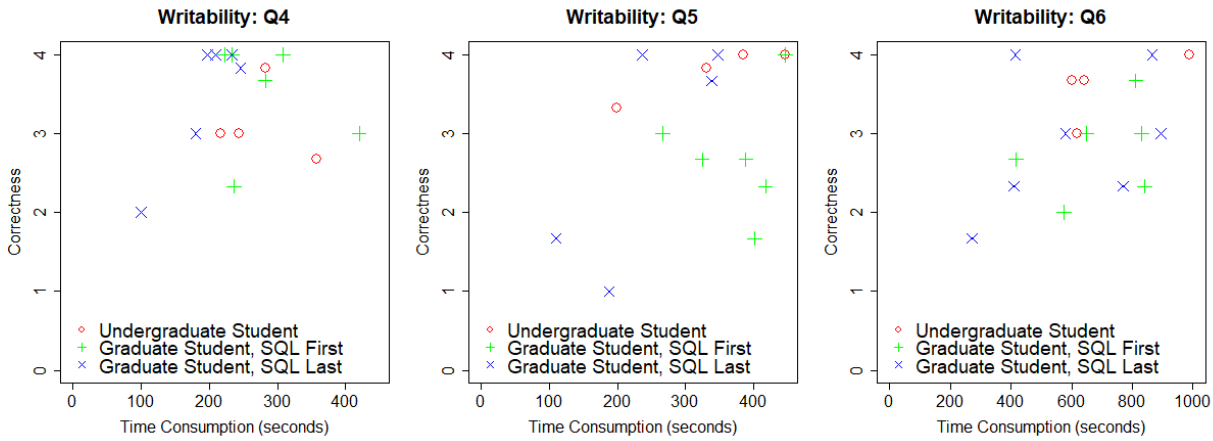

Figure 4.5: Writability Questions: Time Consumption vs. Correctness, with SQL

in SQL$^{\text{path}}$. This is a very interesting observation and we will discuss it later when we start analyzing each question individually.

Going back to our previous discussion, SQL$^{\text{path}}$ is claimed by researchers to have two advantages over traditional SQL. First, by providing a more simplified syntax together with the compressed notation of "attribute path", SQL$^{\text{path}}$ is expected to be easier to work with, hence reduce the time needed to think the question through as well as actually write down the query. Furthermore, this shall also make reading and understanding queries written by others easier. Second, with its simplified syntax and the ability to hide explicit reference to foreign keys for table joins, it shall also be less likely for users to make mistakes working with SQL$^{\text{path}}$ questions, especially for complex conjunctive queries where multiple table joins and complicated constraint checking are required. Here we perform our evaluation, analysis, and comparison of SQL and SQL$^{\text{path}}$ based on these two metrics.

4.4.2 Time Consumption Analysis

Recall that our Null Hypotheses 1 and 3 aimed at evaluating the time consumption difference between SQL and SQL$^{\text{path}}$ for readability and writability questions. If Null Hypothesis 1 held, it should take a participant the same amount of time to read, understand a database query written in SQL$^{\text{path}}$ and translate it into English. Similarly, if Null Hypothesis 3 held, a participant should consume the same amount of time to design and write a database query with SQL$^{\text{path}}$. This can be analyzed by comparing the recorded time consumption of the

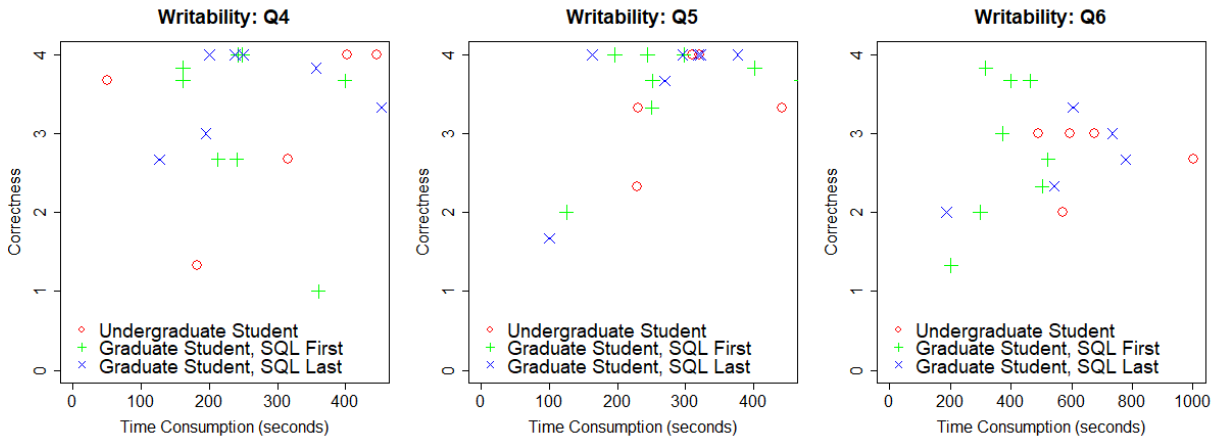

Figure 4.6: Writability Questions: Time Consumption vs. Correctness, with SQL$^{\text{path}}$

questions. For this analysis we discard the data collected for undergraduate students, as we are only interested in the behavioural difference of the same student when working with SQL and SQL$^{\text{path}}$.

For each question, we define a participant's time consumption difference as his consumed time working with SQL minus his consumed time working with SQL$^{\text{path}}$, in seconds. Therefore, if the final value is positive, then the participant spent less time with SQL$^{\text{path}}$ in the experiment. Furthermore, the larger the value is, the faster he was when working with SQL$^{\text{path}}$, which in turn indicates better efficiency and ease to use of SQL$^{\text{path}}$. The box-plots of all graduate participants' time consumption differences for each readability question and writability question are shown at the top part of Figure 4.7 and Figure 4.8, respectively.

However, one may suspect that since graduate participants worked on both SQL and SQL$^{\text{path}}$ questions, they might perform better on the second set of questions to work with, as these questions were very similar to their counterparts in the first part, which would in turn reduce participants' time consumption. To answer this, we further split graduate participants into two groups, one worked with SQL first and the other worked with SQL$^{\text{path}}$ first. The group split results are plotted at the bottom part of Figure 4.7 and Figure 4.8. For all plots, the portion above the red line (0) represents lower time consumption and better efficiency for SQL$^{\text{path}}$.

Readability Questions

As we can see from the box-plots, participants are significantly more efficient with SQL^{path}, when working on question 2 and 3, while question 1 is the only one where time consumption for SQL and SQL^{path} is very close. This may be explained due to the fact that question 1 is the easiest of all six questions. Going back to Table 4.2, we can see that the average time consumption for question 1 across all graduate participants is less than 2 minutes and majority of them have perfect or near perfect correctness marks. Therefore, it may be too simple to provide any meaningful comparison result as participants can come to the solution right away and they are highly unlikely to make any mistake. On the other hand, it's quite clear that for any non-trivial questions involving conjunctive and alike queries, SQL^{path} takes less time and is more efficient to work with.

One may argue that as SQL^{path} has a much simpler syntax, it shall benefit more from the familiarity factor when being worked with later. This may be the reason of its time consumption advantage we observed. From the bottom part of Figure 4.7, we find that SQL^{path}'s time consumption advantage is more noticeable when worked with later, as the box on the left is always higher than the box on the right. This confirms that participants indeed perform better on their second set of questions. But for more complex question such as 2 and 3, SQL^{path} still beats SQL when worked with first.

Overall, the observed results tend to reject our Null Hypothesis 1, as SQL^{path} is shown to be noticeably easier and faster to read and understand. To prove this statistically, we apply Wilcoxon signed rank test to analyze the time consumption difference for all graduate students, as our data is paired, randomly collected from the same population, and measured on an interval scale where the underlying distribution is unknown. When we use the alternative hypothesis that the time consumption of SQL^{path} is less than SQL, the obtained p-values for the three readability questions become 0.5, 0.0027, and 0.0034. With the commonly used probability threshold of 0.05, the Null Hypothesis 1 can be rejected for question 2 and 3 where reading and comprehending SQL^{path} is significantly less time consuming and more efficient.

For question 1, if we switch the alternative hypothesis to that the time consumption of SQL^{path} is greater than SQL, then the obtained p-value becomes 0.5227. This is statistically insignificant and cannot be used to reject our null hypothesis. All of these results prove that for readability questions, SQL^{path} is at least as efficient as SQL, and significantly more efficient when dealing with more complex problems.

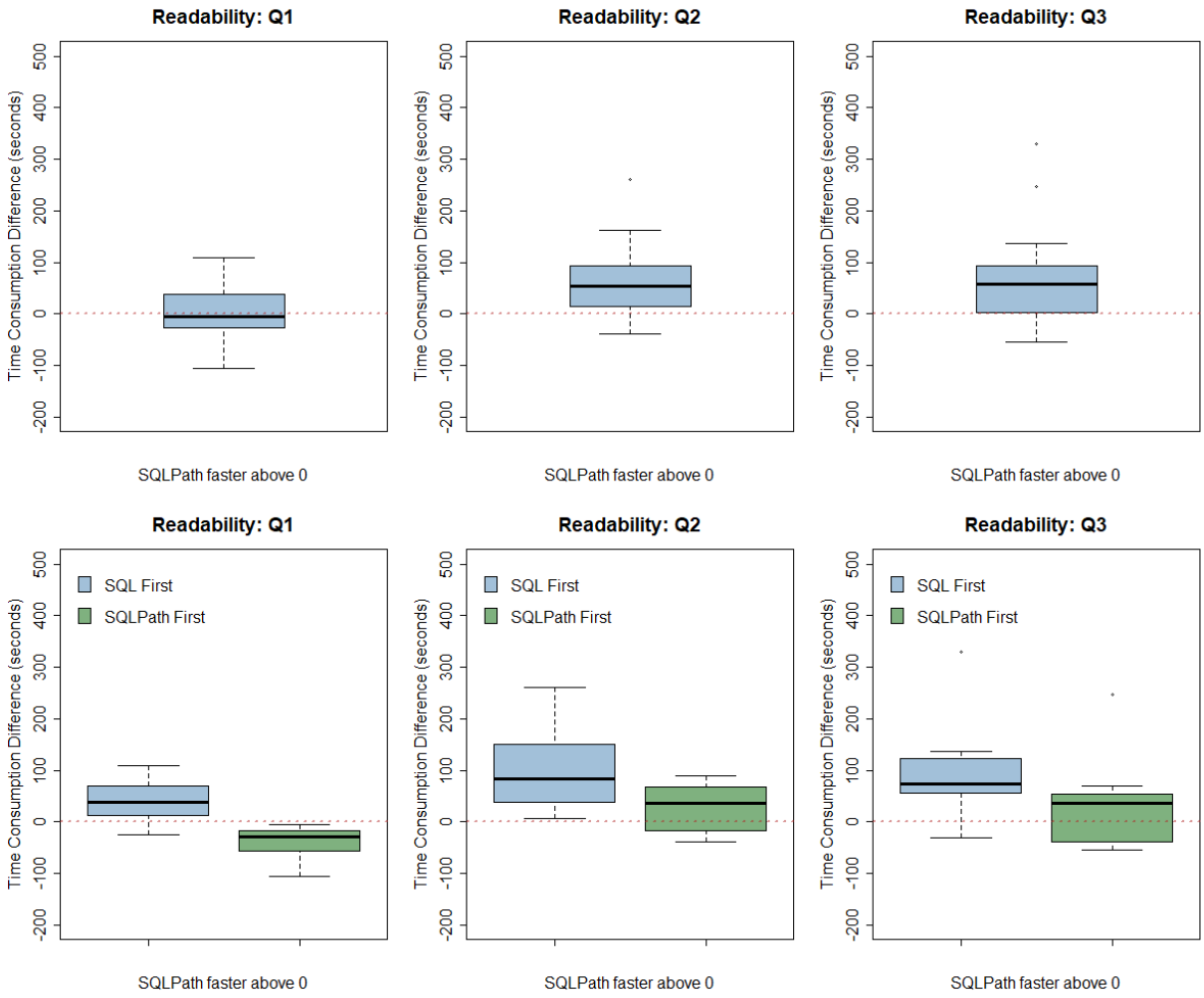

Figure 4.7: Time Consumption Difference: Graduate Students, Readability Questions

Writability Questions

On the other hand, for writability questions, it takes participants significantly less time with SQL$^{\text{path}}$ on question 5. There is also a quite noticeable advantage of SQL$^{\text{path}}$ for question 4 and 6. When we split the participants by order of query language to work with, similar to our previous observation for readability questions, the advantage of SQL$^{\text{path}}$ gets mitigated when worked with first. This time, question 5 become the only one where

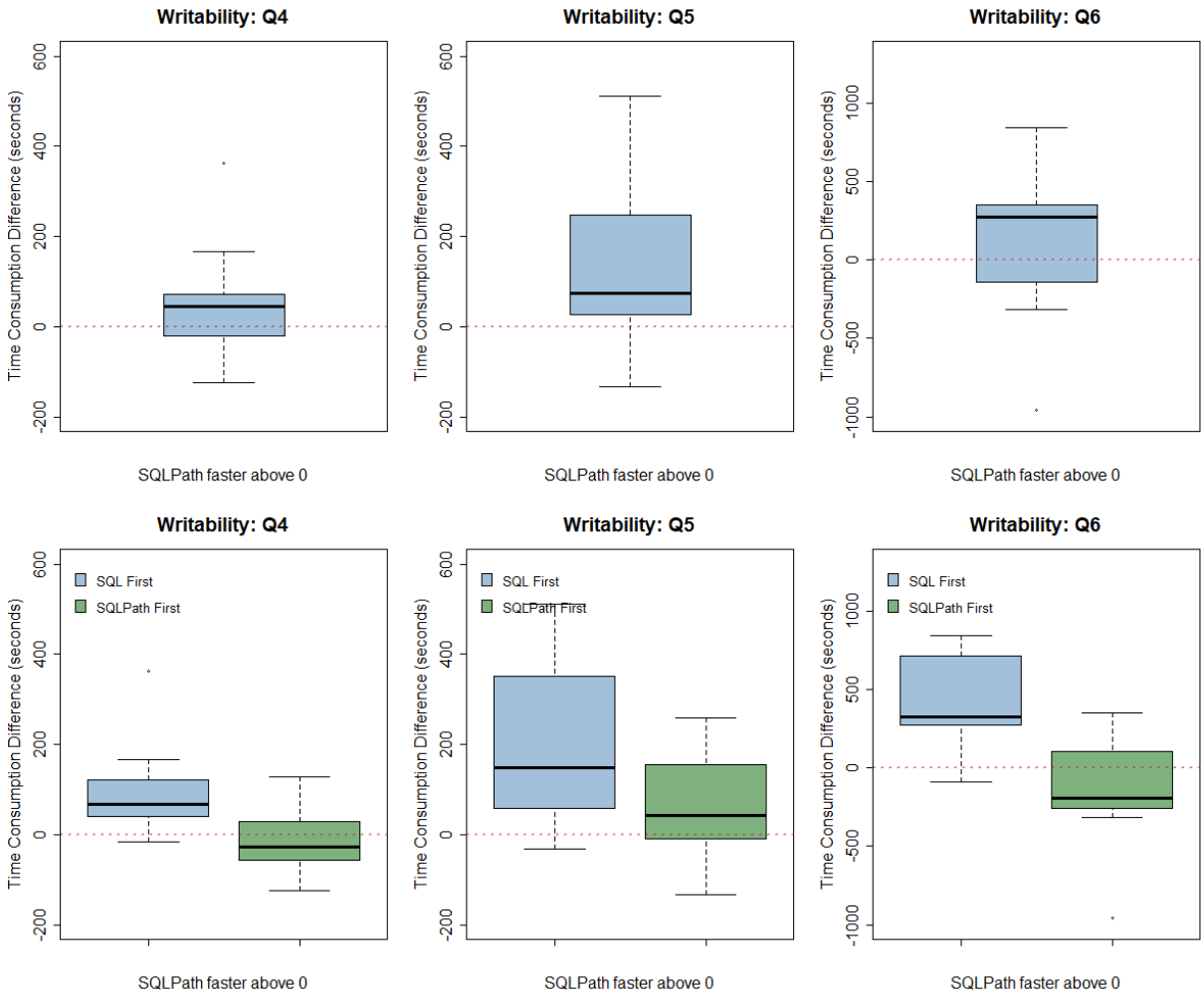

Figure 4.8: Time Consumption Difference: Graduate Students, Writability Questions

SQL$^{\text{path}}$ is clearly faster even when worked with before SQL. The time difference slightly favors SQL for question 4, but SQL has very significant advantage over SQL$^{\text{path}}$ for question 6 when worked with after it.

We suspect that the unique observation of question 6 in Figure 4.8 is due to its extra complexity. Being the hardest question in this experiment with complicated and most number of constraints, it is more time consuming to read and understand the question itself and comprehend the logic behind it, thus more time can be saved working on it the

second time. Result data in Table 4.4 backs this observation as the average time consumed on question 6 is significantly longer than any other question.

Overall, from our observation, we can conclude that SQL^{path} is more efficient to work with for designing and writing conjunctive database queries. After applying Wilcoxon signed rank test again, with alternative hypothesis that the time consumption of SQL^{path} is less, the obtained p-values are 0.0677, 0.0027, and 0.0820 for the three writability questions. With the cutoff threshold of 0.05, our Null Hypothesis 3 can only be rejected for question 2. If we revert the alternative hypothesis to SQL's time consumption was less, the p-values for question 4 and 6 become 0.9397 and 0.9263. This result states that our null hypothesis holds for most of the writability questions, or in another word, SQL^{path} query is not slower to design and write, while in some special case it can even be faster. Statistically speaking, it is more likely that writing SQL^{path} queries is faster as its associated p-values are much smaller. This efficiency advantage become bigger when users are familiar with the problem domain, i.e., worked with the same questions in SQL beforehand. For queries that are excessively more complicated, this advantage may not be seen when working with unfamiliar problems, since a user have to spend time understanding the problem domain as well as the concept of SQL^{path}, which may be too overwhelming.

Combined with our previous observation, SQL^{path} is equal or more efficient to work with, for both readability and writability questions, depending on the complexity level. The next question we want to answer is whether it can improve users' correctness level when reading and writing conjunctive and alike queries. Recall from our earlier definition, correctness can be measured by marks rewarded, as well as detection of syntax errors and purposely shortened solutions.

4.4.3 Correctness Analysis

We are more interested in the correctness marks of the participants' solutions, as it can be used to verify the validity of Null Hypothesis 2 and 4. Following the same approach we applied with our time consumption analysis, we define each graduate participant's correctness difference as the correctness marks of SQL^{path} minus the correctness marks of SQL for each question. Similarly, a positive difference value is an indicator that working with SQL^{path} produces more accurate answer for a question, and the greater the value is, the more accurate the SQL^{path} answer is. The data of undergraduate students is also excluded here, since we only want to analyze the correctness performance of the same participant on the same question, to remove external influence.

The box-plots of all graduate students' correctness difference for readability questions

are shown at the top part in Figure 4.9, while the plots of writability question correctness difference are shown in Figure 4.10. We also split the participants into groups based on the sequence of query language they worked with, and the box-plots of the grouped results are shown at the bottom part of Figure 4.9 and Figure 4.10.

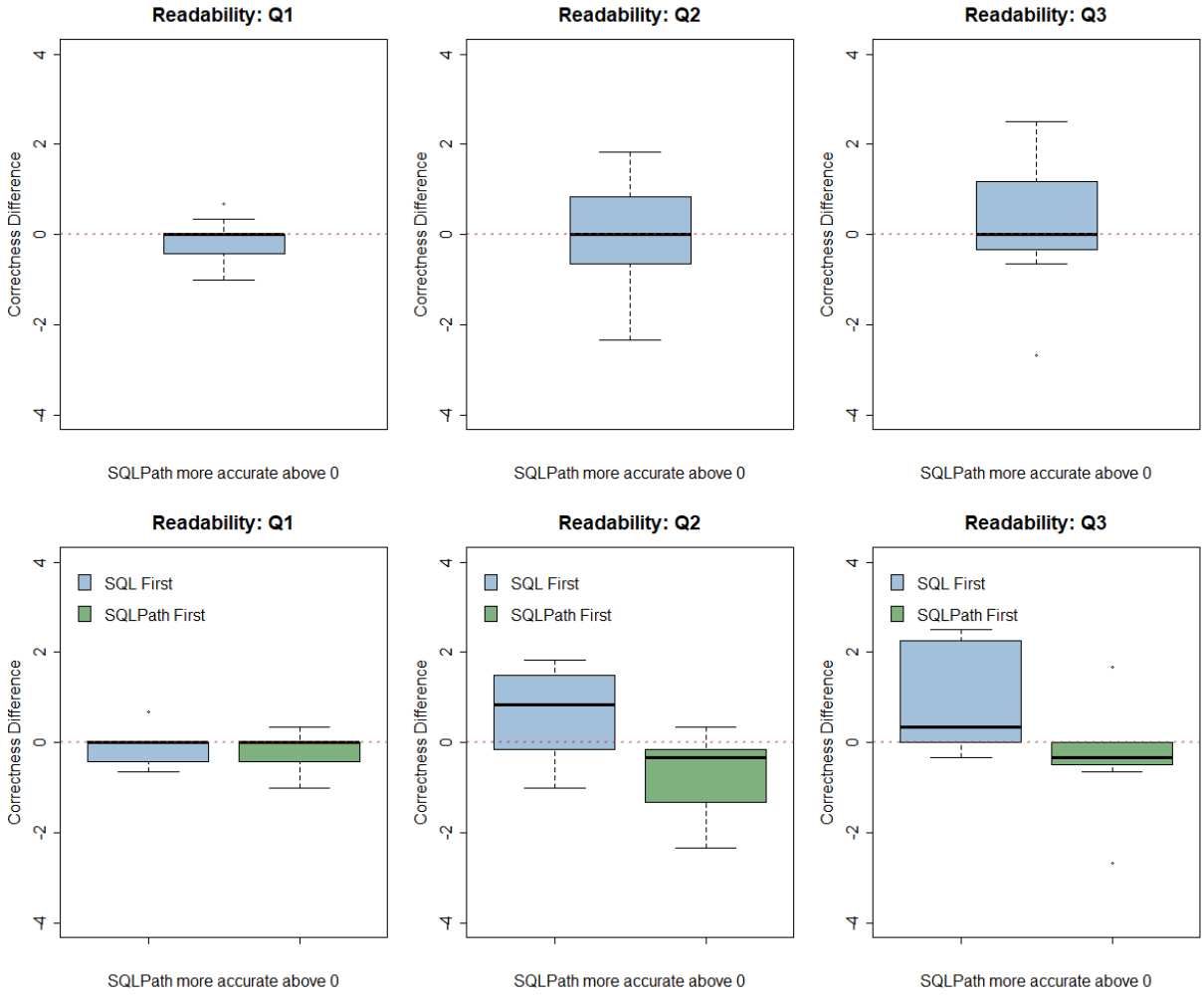

Figure 4.9: Correctness Difference: Graduate Students, Readability Questions

Readability Questions

Overall, the correctness difference between SQL and SQL$^{\text{path}}$ is relatively smaller. From Figure 4.9, SQL$^{\text{path}}$ has no clear edge over SQL for any of the readability questions. In fact, the result is very mixed. It is slightly less accurate for question 1, more or less the same for question 2, and slightly more accurate for question 3. However, none of the correctness difference is significant. Overall, this mixed result does not provide us any evidence indicating whether a participant tends to make more or less mistakes when reading queries written in SQL$^{\text{path}}$, when compared to SQL.

From the grouped plots, the correctness advantage of SQL$^{\text{path}}$ is only observed when being worked with after SQL. In fact, for question 2, its performance is the exact opposite when worked with before SQL. Question 3 also displays a similar pattern. Additionally, for question 1, SQL$^{\text{path}}$ actually performs worse in both cases, although slightly better when worked with after SQL. This observation confirms the belief we had in previous discussion, that SQL$^{\text{path}}$ with its added feature would not help nor hinder one's understanding of a conjunctive or alike query expression, where the correctness difference is mostly due to taking advantage of the similarity when working on the same questions repetitively. Moreover, for extremely simple questions such as question 1, it may even cause slight trouble to a first time SQL$^{\text{path}}$ user. When the SQL query for a question itself is very straightforward that most participants can arrive at the correct solution right away, the unfamiliarity with SQL$^{\text{path}}$'s syntax and added features may prompts undesired mistakes.

Similar to our time consumption analysis, we apply Wilcoxon signed rank test to our correctness data with the alternative hypothesis that SQL$^{\text{path}}$ queries can be understood more accurately first. The correctness difference between SQL$^{\text{path}}$ and SQL solutions are used for the test and the obtained p-values are 0.8968, 0.4624 and 0.2376 for question 1, 2, and 3. If we reverse the alternative hypothesis to assume SQL solutions are more accurate, then the p-values become 0.1307, 0.5624, and 0.7927 respectively. Therefore, Null Hypothesis 2 cannot be rejected as none of the p-values are less than the cutoff threshold of 0.05.

This result leads us to the conclusion that queries written with SQL$^{\text{path}}$, although are easier and faster to read (shown in our time consumption analysis), will not help people to understand better, i.e. the likelihood of a participant to make any mistakes when reading and translating a database query written in SQL$^{\text{path}}$ to English more or less stays the same as the queries are written in SQL. This conclusion yields two results. First, the addition of "attribute path" will not introduce any confusion to users given they understand the concept. Second, as a well defined and established query language, SQL itself is already clear to understand, although its verboseness may require more time to read through.

Writability Questions

From the plots in Figure 4.10, for writability questions, SQL$^{\text{path}}$ has clear edge over SQL for question 5. It is slightly more accurate for question 4, and slightly less accurate for question 6. Similar to readability questions, this mixed result does not give us any proof of SQL$^{\text{path}}$ being more or less accurate when compared to SQL. When we apply Wilcoxon signed rank test to the data with the alternative hypothesis that SQL$^{\text{path}}$ solutions are more accurate, the obtained p-values are 0.3328, 0.0081, and 0.6392 for questions 4, 5, and 6. With the cutoff threshold of 0.05, the Null Hypothesis 4 can only be rejected for question 5. For questions 4 and 6, after reversing the alternative hypothesis, p-values become 0.6954 and 0.3908. The change in correctness in these two questions can not draw any conclusion that is statistically significant enough. Although SQL$^{\text{path}}$ is only significantly better in one question, it is not shown to be statistically worse than SQL.

The observation for writability questions makes us suspect that SQL$^{\text{path}}$ may be easier to design and write for conjunctive queries that are not too complicated, where the challenge is not to understand the question and come up with the logic, but rather to capture all the constraints and write them down completely without making any minor mistakes. This can also explain the clear advantage of SQL$^{\text{path}}$ for question 5. As it has multiple constraints involving table joins that are on the same level without any deep nesting, participants are more likely to make mistakes of missing matching columns in table joins, which is hidden by SQL$^{\text{path}}$'s use of "OID" and "attribute path".

Question 6 is different as it contains deeper level table joins and nested constraints. This may in turn cause trouble to first time SQL$^{\text{path}}$ users as the thinking process with "attribute path" on nested joins and constraints is rather counter intuitive for someone with prior SQL experience. For example, to save lines one may need to start from terminal tables (with reference to but not referenced by other tables) or shared tables (referring or referred by multiple other tables). With SQL, the thinking process almost always start from the table where entities are going to be selected. Hence the most challenging and error prone points for this type of questions is in regard to the logic, rather than writing all of the constraints down completely and correctly. SQL$^{\text{path}}$, as shown here, may not be novice friendly enough on that front.

Although we suspect users with more experience of SQL$^{\text{path}}$ may grasp its intuition better and can then take advantage of the power of "attribute path" to deal with more complex queries more naturally, we don't have enough data here to validate this assumption. In this study we are only aiming to gain some overall insight into SQL$^{\text{path}}$'s performance. It can become a future research direction to look into the performance difference of SQL$^{\text{path}}$ on database queries with different number or depth of constraints, as well as on study

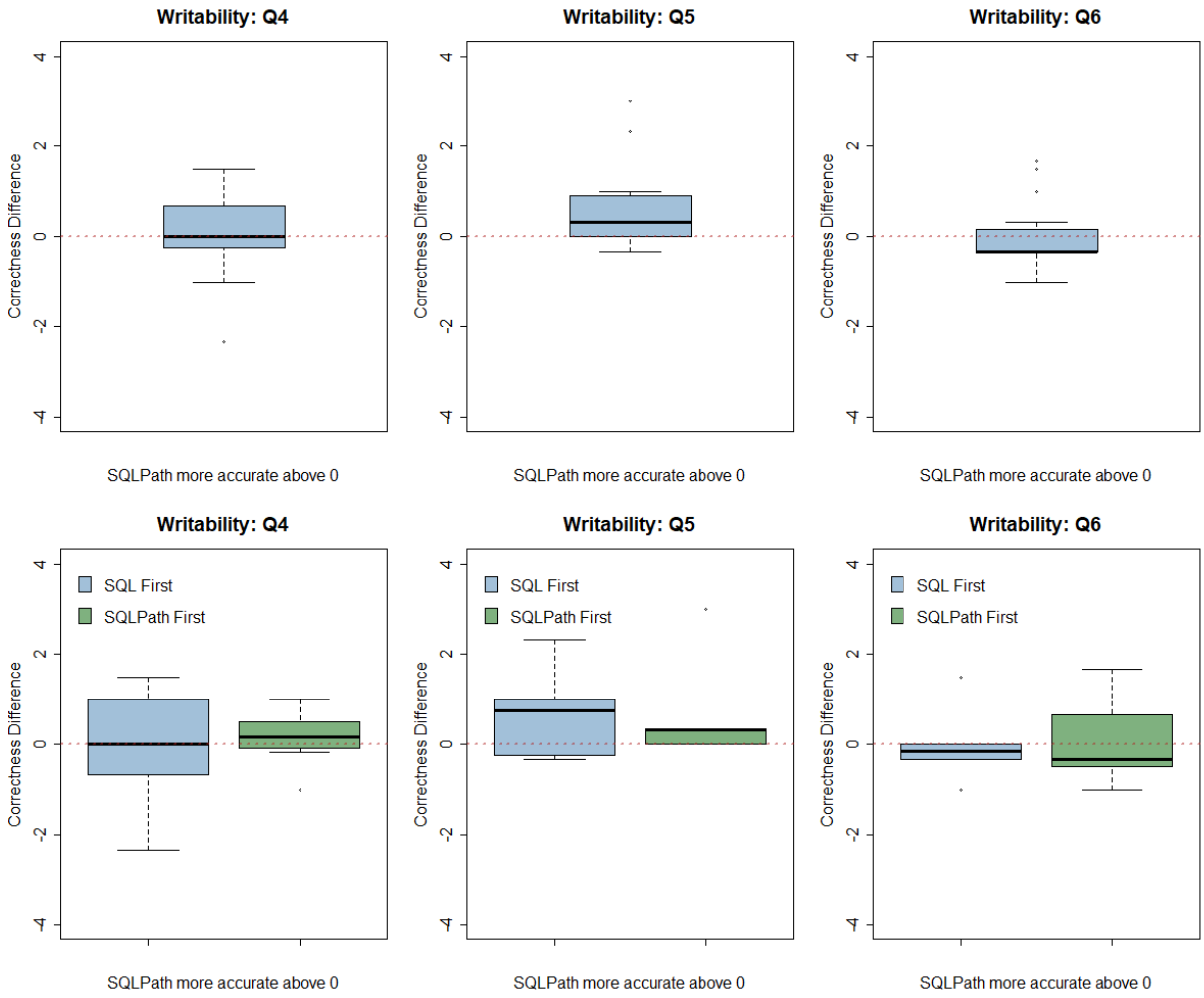

Figure 4.10: Correctness Difference: Graduate Students, Writability Questions

population of different experience level of both SQL and SQL^{path}.

For grouped box-plots, writability questions still yield mixed result. The result for question 5 confirms that for queries with multiple same level constraints, the ability to avoid writing explicit table joins is less error prone and a deciding factor of the correctness difference. Combining with the result for question 4, we can see that SQL^{path} is more accurate to use for database queries, especially conjunctive queries that are not too complicated. For the more difficult question 6, SQL^{path} answers are always less accurate. Moreover, when

worked with first, the correctness difference has a bigger gap and wider spread, which may reflect that different participants were trying to come up with solutions from very different perspectives without thinking through the question in SQL idioms first. When worked with after SQL, participants might have tried an approach that was similarly to what they did with SQL. This result leads us to believe that the lack of experience with SQL$^{\text{path}}$, combined with its unique thinking process may have contributed to its worse performance in writing more complicated conjunctive queries with deeper and nested table joins and associated constraints.

In conclusion, we cannot reject our Null Hypothesis 4, as for writability questions, SQL$^{\text{path}}$ is not better than SQL when correctness of participants' solutions is considered.

4.4.4 Syntax and Inconvenience

Recall for query writing questions we also recorded participants' syntax errors and inconvenient solutions. This can help us discover whether the syntax of SQL$^{\text{path}}$ can easily be understood for first time users. We counted number of solutions containing syntax errors in Table 4.3 and Table 4.4. We deem a solution to contain syntax errors if any of the three markers noticed one while they were marking. We find that solutions written in SQL$^{\text{path}}$ in general contain more syntax errors: 9, 6, and 7 syntax errors were detected for SQL while 8, 10, 14 syntax errors were detected for SQL$^{\text{path}}$, for question 4, 5, and 6 respectively.

Recall that majority of our study population had prior experience with SQL, this observation is actually expected because of the novelty of SQL$^{\text{path}}$'s added feature. By scanning the solutions containing syntax errors, we find that some of the participants tried to integrate SQL's table joins into SQL$^{\text{path}}$, due to the lack of understanding of its "attribute path". Moreover, unlike SQL, the number of syntax errors increases as the complexity of the query increases for SQL$^{\text{path}}$, as more tables are needed to be joined and the attributes' "chaining" gets more complicated. In conclusion, we suspect that although SQL$^{\text{path}}$ has a rather simplistic syntax, it can be quite challenging for new users.

On the other hand, there are 4 inconvenience solutions for SQL and 2 inconvenience solutions for SQL$^{\text{path}}$. This may suggest that SQL$^{\text{path}}$ could be more convenient to work with when compared against SQL, since the usage of "attribute path" can save a lot of manual work when multiple tables are joined, especially in nested sub-queries. However, the sample size may be too small to have any statistical meaning. Overall, we don't find any of the two query languages can be regarded as inconvenient for our experiment.

4.4.5 Other Explanatory Variables

As discussed earlier in the study design section, we have collected data on other explanatory variables: participants' first language, prior experience with SQL and education level. For prior experience, we can only compare it for undergraduate students since all graduate students have worked with SQL before. Since we have only recruited 9 undergraduate participants, the sample size is too small after splitting into two groups to draw any meaningful conclusion. A further experiment with a larger sample size will be required to look into the influence of this as an explanatory variable.

Native vs. Non-Native English Speaker

After grouping all participants by their first language, the time consumption and correctness performance for SQL and SQL^{path} is shown in Figure 4.11 for readability questions and Figure 4.12 for writability questions. From the figures, native and non-native English speakers perform very differently. For readability questions, there is no clear indication whether SQL^{path} is more efficient or accurate than SQL for native speakers. They are faster and more accurate with SQL^{path} for question 3 but slower and less accurate for question 1 and 2. On the other hand, non-native speakers are in general faster for all three questions, and noticeably more accurate for question 2 and 3 with SQL^{path}. Question 1 is the simplest question in the entire experiment hence the familiarity of SQL and unfamiliarity of SQL^{path} may play a bigger role than comprehension of the logic itself. But in general, for more complex questions involving reading comprehension and translating to English, SQL^{path} appears to be more friendly towards non-native speakers as its usage leads to their improvement in efficiency and accuracy, and the performance of non-native speakers is overall better than their native counterparts.

For writability questions, similar performance behavior can be observed. Regarding correctness, native speakers still yield mixed results, but noticeably faster with SQL^{path} across the board. Non-native speakers on the other hand are more efficient on all three questions with SQL^{path}. Their correctness performance with SQL^{path} is the same for question 6, but significantly better for questions 4 and 5. Overall, SQL^{path} seems to be more friendly towards non native speakers for writability problems. Its advantage over SQL is more appreciated for writing database queries than reading database queries for native speakers. However, as our sample size is relatively small, we suggest this observation to be more of a guideline than conclusion.

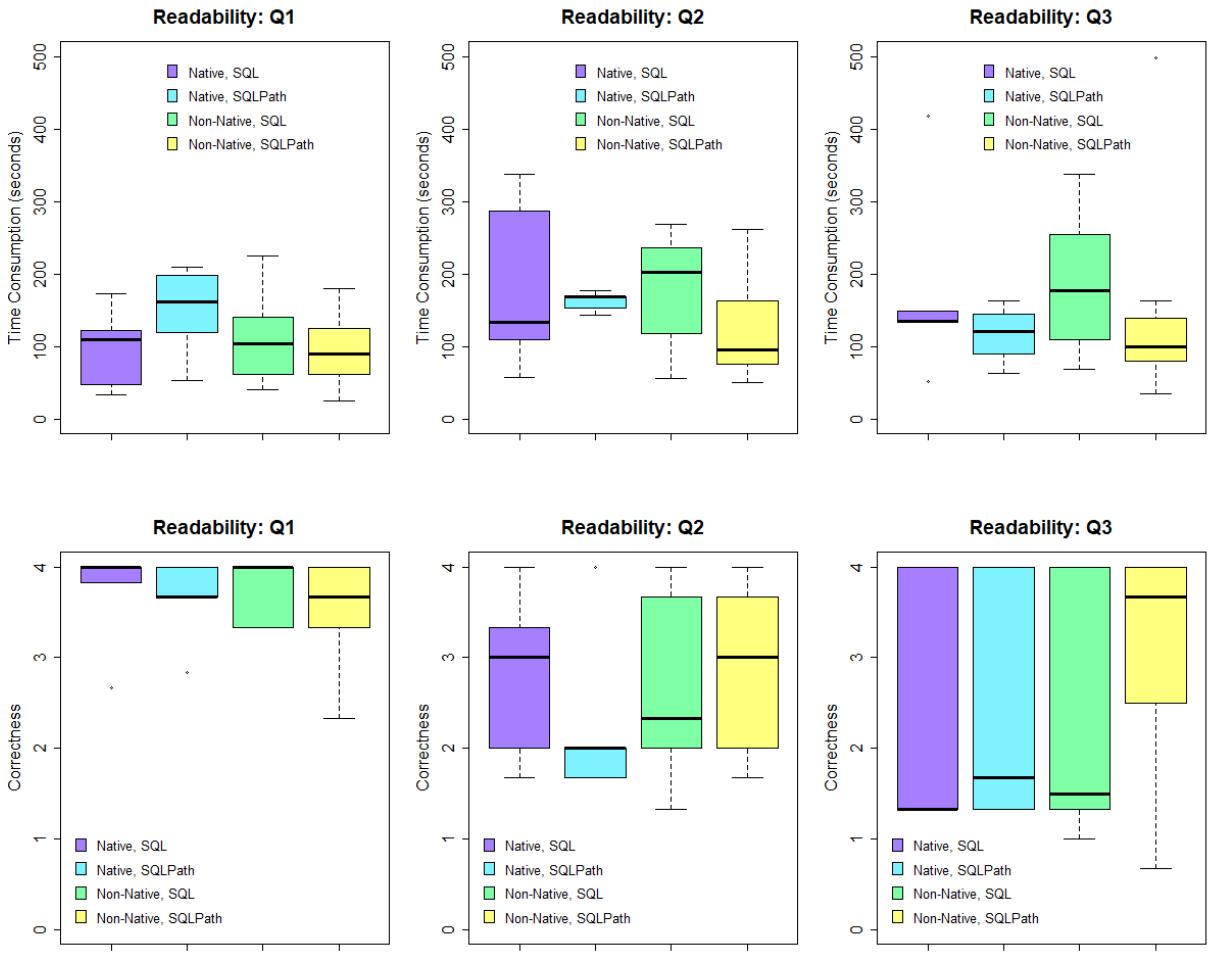

Figure 4.11: Native English Speaker vs. Non-Native English Speaker, Readability Questions

Undergraduate Students vs. Graduate Students

We also analyze the influence of different education levels by splitting the results into groups of undergraduate participants and graduate participants. The grouped box-plots are shown in Figure 4.13 for readability questions and Figure 4.14 for writability questions.

For readability questions, the change of time consumption and correctness moving from

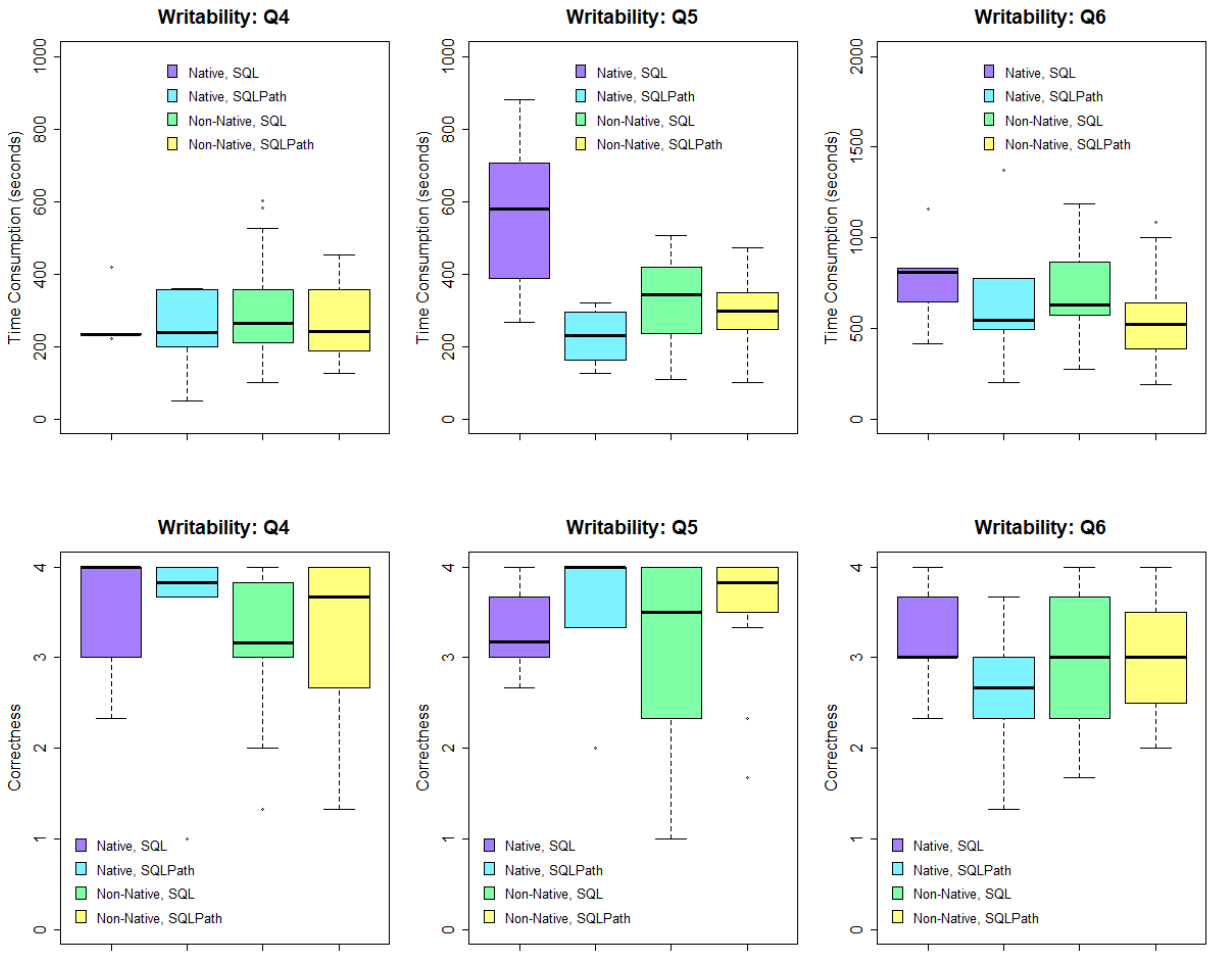

Figure 4.12: Native English Speaker vs. Non-Native English Speaker, Writability Questions

SQL to SQL$^{\text{path}}$ for the same question is almost always mirrored between undergraduate and graduate students. The only outlier is question 2, where graduate students are faster with SQL$^{\text{path}}$ while no significant time consumption difference can be observed for undergraduate students. Overall, this result suggests that the difference of participants' education level does not affect ones ability to adopt and work with SQL$^{\text{path}}$. This conclusion can also be backed by the scatterplots we drew earlier in Figure 4.3 and Figure 4.4, where no clear relationship between education level could be seen.

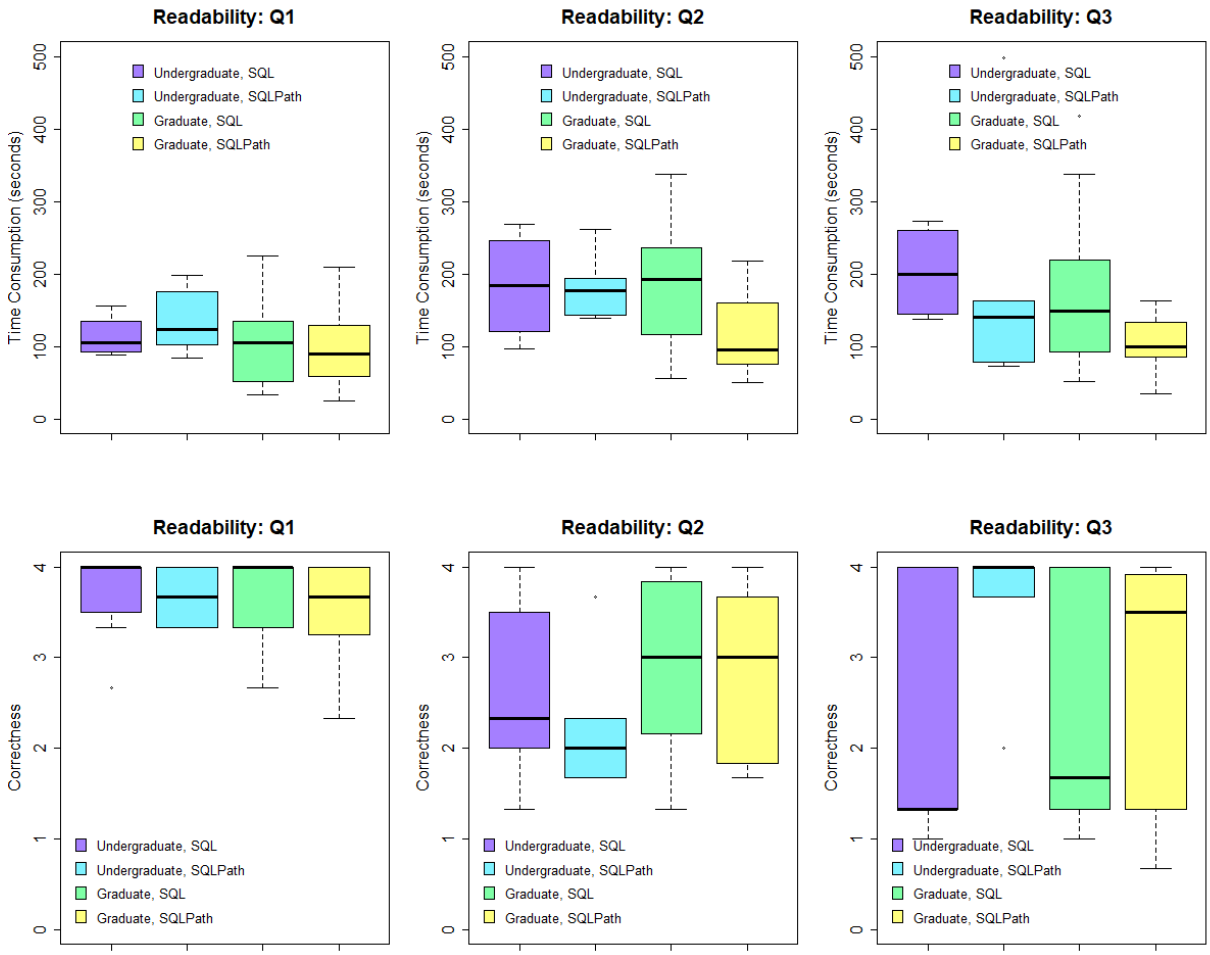

Figure 4.13: Undergraduate vs. Graduate Students, Readability Questions

For writability questions, the accuracy of SQL and SQL^{path} solutions is very close for both undergraduate and graduate students, hence we can not arrive at any conclusion. The trend of efficiency improvement is the same for all participants, but the reduction of time consumption is more noticeable for graduate students. This observation suggests that graduate students are faster at writing SQL^{path} queries. However, generally speaking, we do not think education level is a major factor deciding one's ability to learn and use SQL^{path}.

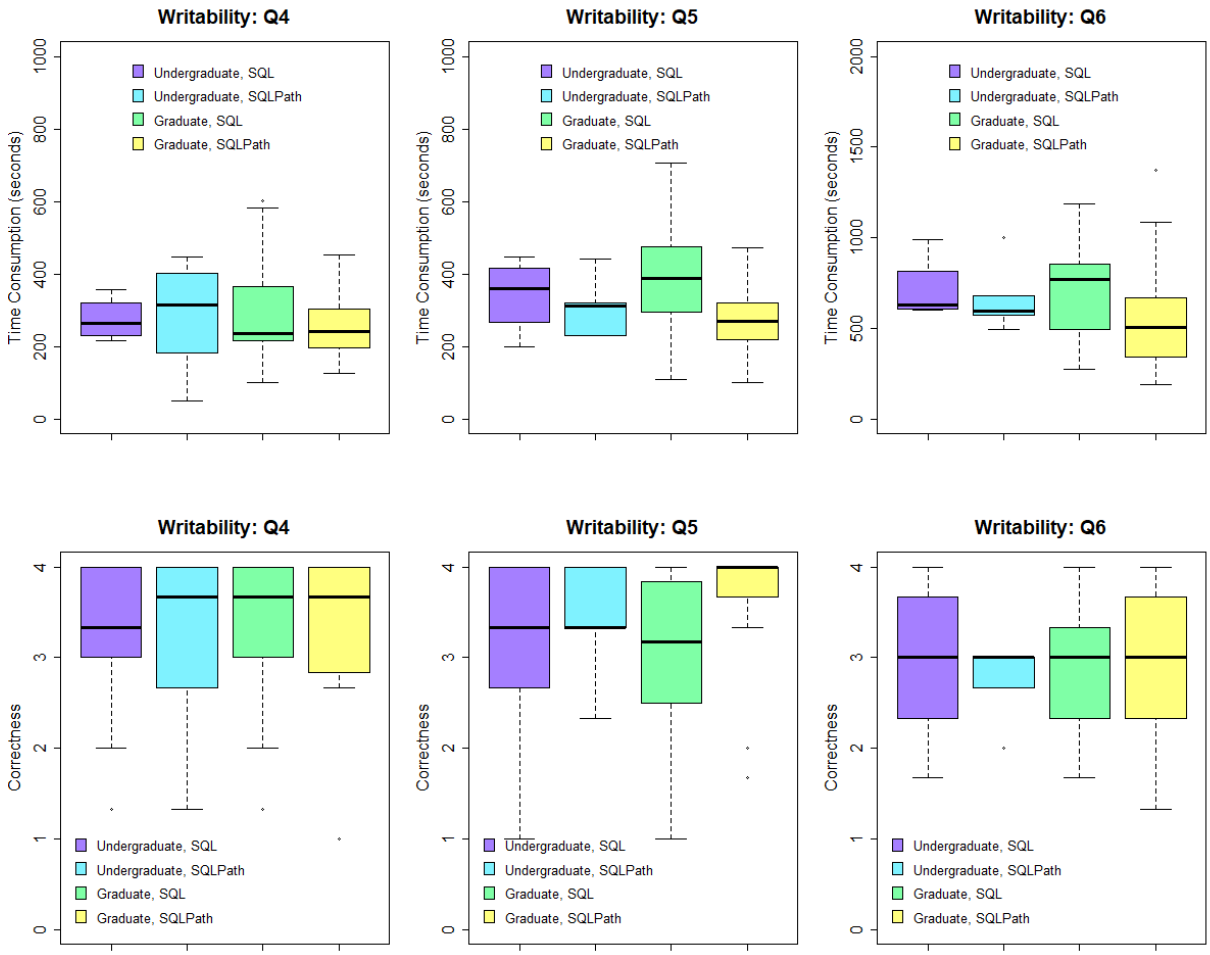

Figure 4.14: Undergraduate vs. Graduate Students, Writability Questions

Overall, although some of the explanatory variables may somewhat affect the understandability and usability of SQL^{path}, their influence is minimal, and cannot be held against the usage of SQL^{path}. Moreover, due to very limited sample size, the observation and conclusion for these explanatory variables still requires further validation. This can become a future research project where larger and more balanced groups of participants specifically designed for these explanatory variables shall be recruited.

Chapter 5

Conclusion

In this thesis, we introduced Borgida, Toman, and Weddell's work on the abstract relational model and SQL$^{\text{path}}$. We explained and demonstrated how to use the conceptual abstract database schema $\mathcal{ARM}$ to model entities and relationships and how to map traditional relational database tables to abstract tables in $\mathcal{ARM}$. Additionally, we explained how to use SQL$^{\text{path}}$ to design and write conjunctive and alike queries and how the usage of "attribute path" can help reduce the verboseness and redundancy of boolean expressions used in table joins.

To validate the benefit of using the abstract relational model and writing SQL$^{\text{path}}$ queries in real world settings, we proposed four null hypotheses that embodies the time consumption and correctness advantages of SQL$^{\text{path}}$ over SQL, for both query reading and writing problems. To verify the validity of these null hypotheses, we set up an empirical study on selected undergraduate and graduate students at University of Waterloo. We designed the experiment to include questions testing readability and writability of SQL$^{\text{path}}$ on various levels of difficulty, and measured and compared the participants' performance with both time consumption and correctness metrics to gain insight into its overall performance in real world applications.

We presented the results with our statistical analysis in this thesis. Overall, we found that we were able to reject the null hypotheses regarding time consumption improvement, and confirmed that SQL$^{\text{path}}$ takes participants less time to read and write, thanks to its simplistic syntax and ability to hide explicit table joins. Furthermore, this observation was more noticeable for more complicated queries where the query rendered in SQL$^{\text{path}}$ was much shorter and clearer.

On the other hand, correctness wise, for people with prior experience, we found no

evidence that SQL^{path} either helps or hinders one's ability to read and understand database queries, as SQL queries are already well defined and structured. Hence it is not conclusive to suggest SQL^{path} is more accurate for readability questions. This was implied by our statistical analysis as none of the null hypotheses regarding correctness could be rejected by the Wilcoxon signed rank test.

In fact, SQL^{path} yielded mixed correctness results for designing and writing conjunctive queries. From the results, we can draw the conclusion that its benefits and advantages are better appreciated for queries of less complexity but a lot imposed constraints on the same level. For this type of problems, the most common mistakes are writing queries that are missing constraint or foreign key columns for table joins. The ability to hide underlying details with "attribute path" makes SQL^{path} a better tool to handle them. However, for more complicated queries where multi-level constraints and nested table joins are involved, the different thinking process of SQL^{path}'s chained table joins may cause trouble to users with prior SQL knowledge as the intuition is rather different in these scenarios. Our experimental result showed that for this type of problems a user in general performed worse with SQL^{path}. This in turn rejected our correctness null hypothesis for writability questions.

5.1 Future Work

The obtained different results of our experiment for different types of questions for the correctness comparison leads us to believe that SQL^{path}'s added features may be more suitable to handle certain types of queries. We also suspect that by having more experience with SQL^{path}, one may grasp its intuition better to take advantage of its added new features. This shall lead us to some future research directions. For instance, we can compare the correctness difference for queries of different numbers and levels of constraints, as well as towards study population of different experience level on both SQL and SQL^{path}.

Furthermore, although the usage of "attribute path" is expected to reduce database queries' verboseness and complexity, the actual extent of improvement is not quantified. It would be interesting to collect SQL queries from real world software applications and compare the line count or boolean expression count against equivalent queries expressed in SQL^{path}.

On the other hand, in our empirical study we only evaluated the usage of the abstract relational model and SQL^{path} mostly at solving conjunctive query focused questions. In fact, they can be used to work with other query types, too. One can design more comprehensive experiments to measure its readability and writability performance and compare

its difference between different query types. This can lead us to find out the most suitable query types for these tools. In addition, we can experiment with more complex database schemata.

Moreover, although we provided a systematic approach to map relational database schema to abstract relational database schema, its validity is not formally proved. We are expected to present a formal proof for both forward and backward mapping in future work.

Last but not least, the abstract relational model and SQL^{path} have only been used in theory. Although systematic approaches to map the relational model to the abstract relational model and compile SQL^{path} to concrete SQL queries are proposed, they are not actually implemented in any of today's RDBMS. It's an interesting direction to build actual compilers that can convert abstract table declaration, referring expression type assignments, and SQL^{path} queries to standard SQL programs, either as a standalone application or plugin to some of the major relational database engines. This can help database developers to get their hands on SQL^{path} for real world software applications.

References

[1] ANSI and ISO. *Information Technology – Database languages – SQL*. Switzerland, 1st edition, 1986.

[2] ANSI and ISO. *Information Technology – Database languages – SQL*. Switzerland, 5th edition, 2016.

[3] Alexander Borgida, David Toman, and Grant Weddell. On referring expressions in information systems derived from conceptual modelling. In *Conceptual Modeling*, pages 183–197, 2016.

[4] Alexander Borgida, David Toman, and Grant E Weddell. On referring expressions in query answering over first order knowledge bases. In *KR*, pages 319–328, 2016.

[5] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[6] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

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

[8] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[9] Terry Halpin. Modeling of linguistic reference schemes. *International Journal of Information System Modeling and Design (IJISMD)*, 6(4):1–23, 2015.

[10] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys (CSUR)*, 19(3):201–260, 1987.

[11] Vitaliy L. Khizder, David Toman, and Grant Weddell. Reasoning about duplicate elimination with description logic. In *Computational Logic — CL 2000*, pages 1017–1032, 2000.

[12] John Mylopoulos, Philip A. Bernstein, and Harry K. T. Wong. A language facility for designing database-intensive applications. *ACM Trans. Database Syst.*, 5(2):185–207, 1980.

[13] Nicole Schweikardt, Thomas Schwentick, and Luc Segoufin. Database theory: Query languages. In *Algorithms and theory of computation handbook*, pages 19–19. Chapman & Hall/CRC, 2010.

[14] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database System Concepts*. McGraw-Hill New York, 6th edition, 2011.

[15] David Toman and Grant Weddell. On attributes, roles, and dependencies in description logics and the ackermann case of the decision problem. In *In Proceedings of Description Logics, CEUR-WS, vol.49*, pages 76–85, 2001.

[16] David Toman and Grant Weddell. Conjunctive query answering in $\mathcal{CFD}_{nc}$: A ptime description logic with functional constraints and disjointness. In *Australasian Joint Conference on Artificial Intelligence*, pages 350–361, 2013.

[17] David Toman and Grant Weddell. On adding inverse features to the description logic $\mathcal{CFD}^{\forall}_{nc}$. In *PRICAI 2014: Trends in Artificial Intelligence*, pages 587–599, 2014.

[18] David Toman and Grant E Weddell. Applications and extensions of ptime description logics with functional constraints. In *IJCAI*, pages 948–954, 2009.

[19] David Toman and Grant E Weddell. Pushing the $\mathcal{CFD}_{nc}$ envelope. In *Description Logics*, pages 340–351, 2014.