The Utility of the Abstract Relational Model and Attribute Paths in SQL

Weicong Ma[1], C. Maria Keet[2], Wayne Oldford[1], David Toman[1], and Grant Weddell[1]

[1] Cheriton School of Computer Science, University of Waterloo, Canada
{w34ma,rwoldford,david,gweddell}@uwaterloo.ca
[2] Department of Computer Science, University of Cape Town, South Africa
mkeet@cs.uct.ac.za

Abstract. It is well-known that querying information is difficult for domain experts, for they are not familiar with querying actual relational schemata due to the notions of primary and foreign keys and the various ways of representing and storing information in a relational database. To overcome these problems, the Abstract Relational Model and the query language, SQLP, have been proposed. They are the theoretical foundations and ensure that explicit primary and foreign keys are hidden from the user's view and that queries can be expressed more compactly. In this paper we evaluate these theoretical advantages with user studies that compare SQLP to plain SQL as the baseline. The experiments show significant statistical evidence that SQLP indeed requires less time for understanding and authoring queries, with no loss in accuracy. Considering the positive results, we develop a method to reverse engineer legacy relational schemata into abstract relational ones.

1 Introduction

Writing and understanding queries, especially by domain experts, is well-known to have a steep learning curve. Therefore, four strategies have been proposed to alleviate this problem: either SQL is hidden behind a controlled natural language or a visual query language, or the relational or conceptual model is also shown as convenient overview of the database whilst querying in SQL. They have been shown to result in improvements over plain SQL at the database schema layer, demonstrating equal or fewer errors always in less time, and that a "good match of query language and database structure leads to better performance" [10] (see [3, 10] and references therein). A conceptual view (cf. SQL only) also enabled domain experts to invent new queries [4].

The major drawback of conceptual query languages is that most of them support only a subset of the full SQL and therefore have little uptake in industry. Querying with SQL and the relational model (RM) is one step up from the baseline of SQL on the SQL schema, but it still entails the drawback of premature resolution of identification issues (e.g., primary keys). In order to abstract away from that, the notion of *referring expression types* has been proposed [1, 2], which are object identifiers taken from a separate domain, $D_{\texttt{oid}}$, and included in each

table in a schema as a `self` attribute. Each `self` is a single column with a functional dependency to an n-attribute ($n \geq 1$) identifier meaningful to the modeler. With this extension, the RM can be pushed up toward the conceptual layer into an *Abstract Relational Model* (ARM), which makes orthogonal those issues of identification [1, 2], and approaches a conceptual data model through lossless projections (vertical partitioning). Yet, SQL can be retained fully for data manipulation. Moreover, since each `self` is a single column, foreign key joins can always be expressed more compactly as *attribute paths*.

Attribute paths enable seamless path queries that do not require declaring multiple SQL joins manually. *SQL with paths*, SQLP (an extension of full SQL), is such a path query language (extending SQL^{path} [2]) where the foreign key joins are expressed with sequences of dot-separated attributes, which is a longstanding feature of class-based conceptual models [9]. This results in shorter queries and thus fewer chances of making mistakes, and it affords explicit navigation across the ARM to the desired elements. These are advantages in theory. It is not known whether this holds also in practice, especially regarding path query languages. In fact, while multiple path query languages have been proposed, to the best of our knowledge, only PathSQL that collapses left outer joins has been evaluated [7], claiming shorter writing time and fewer errors cf. plain SQL.

We seek to shed light on this issue with, first, a user evaluation that compares SQLP+ARM against the baseline of SQL+RM, testing for both time to write and to comprehend a query and correctness of the written queries. The main outcome of the evaluation is that working with SQLP has been shown to be statistically significantly faster, with the same level of correctness. Therefore, we have devised a novel method that transforms a legacy RM into the richer ARM. This method resolves the issues of identification automatically thanks to the referring expression type assignment inferred during the construction of the ARM, which is identity resolving, and therewith is also capable of making certain implicit constraints of the RM explicit in the ARM, such as disjointness and class subsumption (backward compatibility from SQLP→SQL and ARM→RM is already possible and ARM may be reconstructed in a Description Logic [2]). We also show that the space of SQLP queries remains invariant with respect to vertical partitioning, so that domain experts can use SQLP over a conceptual-like view of an ARM schema while database administrators can view the same schema as tables with many attributes.

In the remainder of the paper, we present the background and the novel method first in Sections 2 and 3 and subsequently the user evaluation in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Background

We begin by introducing the *abstract relational model* (ARM), which is based largely on an earlier version presented in [2], and extended where indicated below. An ARM augments the underlying domain of concrete values assumed in the *relational model* (RM) with an additional abstract domain of entities.

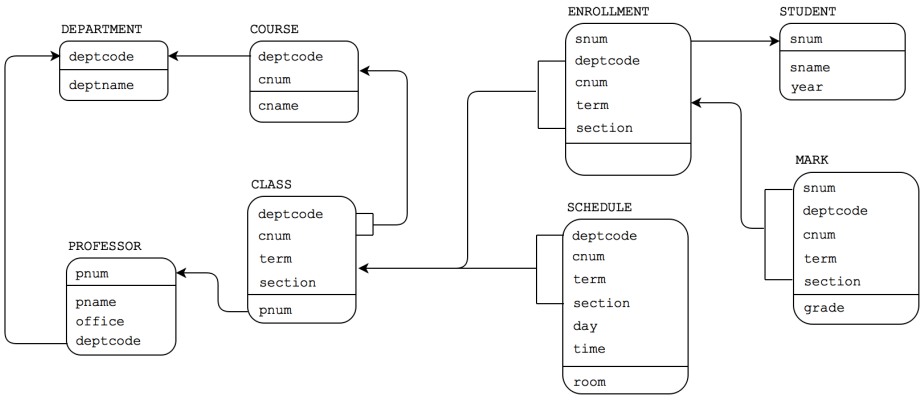

Fig. 1. Course Enrollment as an RM Schema.

Definition 1 (Tables and Constraints) Let TAB, AT, and CD be sets of table names, attribute names that includes `self`, and *concrete domains* (data types), respectively, and let `OID` be an *abstract domain* of *entities (surrogates)*, disjoint from all concrete domains. A *general relational model schema* Σ is a set of *table declarations* of the form[3] `table` T `(`A_1 D_1`, ..., `A_k D_k`, `φ_1`, ..., `φ_ℓ`)`, where $T \in \mathsf{TAB}$, $A_i \in \mathsf{AT}$, $D_i \in (\mathsf{CD} \cup \{\texttt{OID}\})$, and φ_j are *constraints* attached to table T. We write $\textsc{Attrs}(T)$ to denote $\{A_1, \ldots, A_k\}$ and $\textsc{Tables}(\Sigma)$ to denote all table names declared in Σ. A_i is *abstract* if D_i is `OID`, and *concrete* otherwise, and `self` must always be abstract. In addition to attribute declarations "$(A_i\ D_i)$", a variety of constraints φ_j can occur in a table declaration of T, e.g.,

1. (*primary keys*) `primary key (`$A_1, \ldots, A_k$`)`
2. (*foreign keys*) `constraint` N `foreign key (`$A_1, \ldots, A_k$`) references` T
3. (*inheritance*) `isa` T_1
4. (*disjointness constraints*) `disjoint with (`$T_1, \ldots, T_k$`)`
5. (*path functional dependencies*) `pathfd (`$\mathsf{Pf}_1, \ldots, \mathsf{Pf}_k$`)` $\rightarrow$ Pf □

Primary keys and named foreign keys ("N" in item 2, above) are supported by standard SQL. Inheritance and disjointness constraints are only meaningful when `self` occurs as one of the attributes of T and each T_i, and are satisfied when all (resp. no) `self`-value occurring in T occurs as a `self`-value in some (resp. all) T_i in the inheritance (resp. disjointness) cases. Path functional dependencies are a generalization of functional dependencies introduced in [13] that allow *attribute paths* in place of attributes. An attribute path Pf is either `self` or a dot-separated sequence of attribute names excluding `self`, as defined in [2]. A path functional dependency is satisfied when any pair of T-tuples that agree on the value of each Pf_i also agree on the value of Pf. Finally, RM and ARM are obtained by restricting how tables and constraints can be declared:

Definition 2 (RM and ARM) In RM: attributes A_i can only be declared to be *concrete*, and only primary and foreign key constraints are allowed. In ARM,

[3] This is essentially the syntax for SQL's `create table` commands.

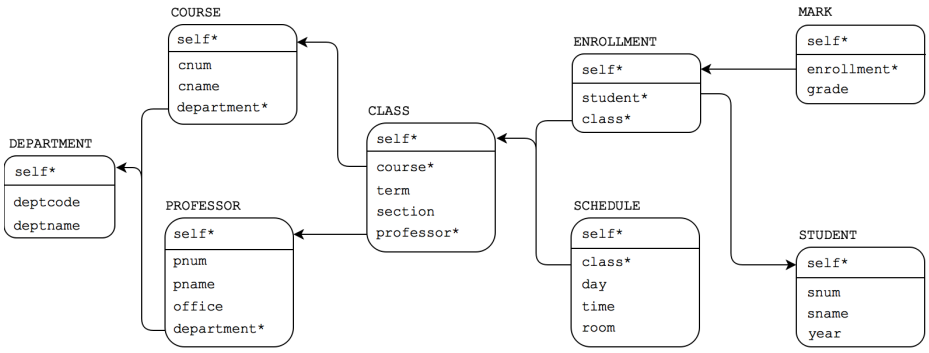

Fig. 2. Course Enrollment as an ARM Schema.

every table has the attribute `self` declared to be its primary key. Consequently, every foreign key constraint must use a single abstract attribute. □

Example 3 A commonly used visualization of RM schemata for a hypothetical course enrollment application is given in Fig. 1: each rectangle is a table, labelled by name, containing the attributes defined on that table. Attributes above the line in a rectangle give the primary key and directed edges between tables show foreign keys. The same visualization approach is used for a counterpart ARM schema in Fig. 2, where abstract attributes are indicated with an '*'. The respective definitions for, e.g., the `CLASS` table are as follows:

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

Logical Implication in Schemas. Given $T \in \mathsf{TAB}(\Sigma)$ and constraint $\varphi \in T$ (possibly not occurring in Σ), we write $\Sigma \models (\varphi \in T)$ to say that φ always holds for T in *any* database over Σ. For example, the requirement of foreign key constraints to be unary in ARM yields the following deduction:

$$\{(\texttt{foreign key}\ (A)\ \texttt{references}\ T_2) \in T_1, (B\ D) \in T_2\} \models (A.B\ D) \in T_1$$

in which the deduced constraint "$(A.B\ D) \in T_1$" states (with slight abuse of notation) that the attribute path "$A.B$" originating in T_1 always ends in D. It is easy to see that the deduction above can be generalized to yield longer paths.

ARM can be formalized in a Description Logic (DL), e.g., the PTIME decidable $\mathcal{CFDI}_{nc}^{\forall-}$ [11], where the problem of deciding when $\Sigma \models (\varphi \in T)$ holds in ARM schemata can be reduced to reasoning about logical consequence [2].

$\langle query \rangle$ ::= `select distinct` x_1`.`Pf_1 [`as` A_1], ..., x_m`.`Pf_m [`as` A_m] $\langle body \rangle$
| $\langle query \rangle$ `union` $\langle query \rangle$
$\langle body \rangle$ ::= `from` T_1 x_1, ..., T_n x_n [`where` $\langle pred \rangle$]
$\langle pred \rangle$ ::= x_1`.`Pf_1 *op* x_2`.`Pf_2 | x`.`Pf_1 *op* c | $\langle pred \rangle$ `and` $\langle pred \rangle$
| $\langle pred \rangle$ `or` $\langle pred \rangle$ | `not` $\langle pred \rangle$ | `exists ( select *` $\langle body \rangle$ `)`

Fig. 3. SQLP (fragment), extended from [2].

SQLP Queries. Since ARM schemata resemble RM schemata, simple revisions to the SQL standard yield SQLP: Fig. 3 shows a relationally complete fragment of the SQL query language grammar modified to allow attribute paths (definition for full SQL is analogous but beyond the limits of this paper). Example 4 illustrates the potential advantages of SQLP.

Example 4 Consider a query over the RM schema of Fig. 1, computing the *names of students who have experienced being taught by professor Alan John.*

```
select distinct s.sname as name
from STUDENT s, ENROLLMENT e, CLASS c, PROFESSOR p
where e.snum = s.snum and e.deptcode = c.deptcode
and e.cnum = c.cnum and e.term = c.term and e.section = c.section
and c.pnum = p.pnum and p.pname = 'Alan John'
```

The same query in SQLP over the corresponding ARM schema (Fig. 2) is:

```
select distinct e.student.sname as name from ENROLLMENT e
where e.class.professor.pname = 'Alan John'
```

Invariance of SQLP Under Vertical Partitioning. A hitherto unnoticed implicit benefit of ARM and SQLP, is SQLP's invariance under vertical partitioning, which we specify explicitly and precisely here.

Definition 5 (Vertical Partition) Let Σ an ARM schema and T(`self OID`, A_1 $D_1, \ldots, A_k$ $D_k, \varphi) \in \Sigma$. We say that an ARM schema Σ' is a *partition of* Σ *(with respect to* T*)* if

$$\begin{aligned}\Sigma' = \Sigma - \{&T(\texttt{self OID}, A_1\ D_1, \ldots, A_k\ D_k, \varphi)\} \\ &\cup \{T(\texttt{self OID}, A_1\ D_1, \ldots, A_i\ D_i, \varphi, \texttt{isa}\ T'), \\ &\quad T'(\texttt{self OID}, A_{i+1}\ D_{i+1}, \ldots, A_k\ D_k, \varphi, \texttt{isa}\ T)\}\end{aligned}$$

where $T' \notin \mathsf{TAB}(\Sigma)$. □

Observe that vertical partitioning constitutes a lossless-join decomposition, and that every instance DB of Σ can be transformed to an instance DB' of Σ' with no loss of information. A repeated application of vertical partitioning ultimately leads to an ARM schema in which all tables are at most binary. This obtains a schema that comes very close to matching class and attribute-based conceptual models: a unary "`self`" table that may be viewed as a class, and binary tables that may in turn be viewed as attributes of the class. Our first result is that, for SQLP queries, answers are invariant with respect to such vertical partitioning:

Proposition 6 Let φ be a SQLP query over an ARM schema Σ, DB an instance of Σ and DB' a corresponding instance of Σ' a partition of Σ. Then $\varphi(DB) = \varphi(DB')$.

This feature allows users the freedom of formulating SQLP queries equivalently over a wide range of ARM schemata, as long as they are related by the "partition of" relation introduced in Definition 5. In particular, domain experts can now use SQLP over a conceptual-like view of a particular ARM schema while SQL programmers can think about the same schema in terms of tables with many attributes (this follows from the bi-directionality of Definition 5).

3 On Deriving ARM Schemata from RM Schemata

To present our second contribution, we need to introduce the notion of *referring types* and *referring type assignments* to tables in ARM that will be constructed by our algorithm together with the ARM schema. They enable one to execute SQLP queries formulated over the created ARM schema translations in [2].

Referring Expression Types and Assignments. To connect ARM and RM schemata, we use the notion of referring expression types from [2]: descriptions of how abstract `OID` values used to identify entities in an ARM schema are represented in a corresponding RM schema. In particular, this entails a *referring expression type assignment* for each T in the ARM schema, denoted $\mathsf{RET}(T)$, with the general form given by

$$T_1 \rightarrow (\mathsf{Pf}_{1,1} = ?, \ldots, \mathsf{Pf}_{1,n_1} = ?)\,;\, \cdots\,;\, T_k \rightarrow (\mathsf{Pf}_{k,1} = ?, \ldots, \mathsf{Pf}_{k,n_k} = ?)$$

where $k > 0$, and each attribute path $\mathsf{Pf}_{i,j}$ is defined on table T_i. Each subexpression separated by *preference indicators* ";" is called a *component* of the RET, and the "$T_i \rightarrow$" part its *guard*. The last component of the RET may not have its guard, which is then inferred to be "$T \rightarrow$". Assigning a RET of this form to each table in the ARM schema, is *identity resolving* if naming issues are sufficiently resolved to enable translating *any* SQLP query over the ARM schema to an equivalent SQL query over its corresponding RM schema [2].

Example 7 The RET for the table `CLASS` in Example 3 is given by

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

stating that a class can be identified by a combination of the four values defined by the indicated paths. This yields the definition of the corresponding `CLASS` table in RM (see Fig. 1).

Observe that this simple kind of RET is still a strict generalization of traditional primary keys due to the possible use of the attribute paths, and that this example relies on this expressiveness. To see why more than one component might be needed, consider where `self`-values in the `PROFESSOR` and `STUDENT` tables could overlap (i.e., if professors could also be students). Were this possible, naming issues could still be resolved by choosing the RET $\texttt{STUDENT} \rightarrow (\texttt{snum} = ?)\,;\, (\texttt{pnum} = ?)$ for `PROFESSOR`. A professor who is also a student would then be identified by their `snum`-value in preference to her `pnum`-value. (This would also require additional attributes to be added to an RM counterpart to the `PROFESSOR` table declaration, attribute `snum` for example; see [1, 2] for details.)

From RM to ARM. In [2], a GenRM procedure is defined that computes an RM schema given an ARM schema and RET assignment as input. We now introduce a complementary GenARM procedure for reverse engineering RM schemata: given a RM schema Σ_1 as input, GenARM modifies Σ_1 to a corresponding ARM schema Σ_2 and computes an RET for Σ_2, for which GenRM(Σ_2, RET) re-obtains Σ_1 (up to names of re-introduced concrete attributes), and for which RET is *identity resolving*, which is a condition defined in [2] ensuring that any SQLP query over Σ_2 can be mapped to an equivalent SQL query over Σ_1.

GenARM modifies Σ and computes RET with the use of a stack S of tables in Tables(Σ), and a *pending assignment* set PA. The latter consists of 4-tuples (T, Pf, T', A) that are used to incrementally compute RET as each table in S is processed. A 4-tuple asserts: *the primary key of T has component $\mathsf{Pf} \circ A$, and depends via foreign key join path Pf, on table T' having attribute A.* Here, "$\circ$" composes attribute paths, in particular: $\mathsf{Pf}_1 \circ \mathsf{Pf}_2$ denotes Pf_1 (resp. Pf_2) if Pf_2 (resp. Pf_1) is `self`, and $\mathsf{Pf}_1 \,.\, \mathsf{Pf}_2$ otherwise. GenARM is defined as follows:

(*initialize S,* PA *and* RET)

1. $S \leftarrow [\,]$, $\mathsf{PA} \leftarrow \emptyset$.
2. For each $T \in$ Tables(Σ) with primary key $(A_1, \ldots, A_m)$:
   2.1. $\mathsf{RET}(T) \leftarrow$ "$T \rightarrow (A_1 = ?, \ldots, A_m = ?)$".
   2.2. Add $(T, \texttt{self}, T, A_i)$ to PA for $1 \leq i \leq m$.
3. Construct a directed graph $G(\text{Tables}(\Sigma), E)$, where E is obtained as follows: for each $T_1 \in$ Tables(Σ) with primary key $(A_1, \ldots, A_m)$, and each foreign key constraint "`foreign key` $(B_1, \ldots, B_n)$ `references` T_2" from T_1: 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.

(*conversion to an* ARM *schema, and refinement of* RET)

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 OID`)" to the definition of T.
   5.3. Let L consist of all tables $T_i \in$ Tables(Σ), where $(A_{i,1}, \ldots, A_{i,k})$ is the primary key of T_i, for which $\{A_{i,1}, \ldots, A_{i,k}\} \neq \{A_1, \ldots, A_m\}$. Add disjointness constraint "`disjoint with` L" to T if L is nonempty.
   5.4. For each $\varphi =$ "`foreign key` $(B_1, \ldots, B_k)$ `references` T_1" from T, where the primary key of T_1 is $(C_1, \ldots, C_k)$, do the following:
      5.4.1. If $\{B_1, \ldots, B_k\} = \{A_1, \ldots, A_m\}$, then replace φ by specialization constraint "`isa` T_1", assign `self` to NA, and proceed to 5.4.3.
      5.4.2. Assign the constraint name of φ to NA, and replace φ by foreign key constraint "`foreign key` (NA) `references` T_1".
      5.4.3. For each B_i, $1 \leq i \leq k$, if $C_i \in$ Attrs(T_1) and if all remaining foreign key constraints φ' for T are free of B_i, then do the following:
         – Remove B_i from Attrs(T).
         – If $B_i \in \{A_1, \ldots, A_m\}$, then for every tuple $t = (T_2, \mathsf{Pf}, T, B_i)$ in PA, for some T_2 and Pf: Replace "$\mathsf{Pf} \circ B_i = ?$" in $\mathsf{RET}(T_2)$ by "$(\mathsf{Pf} \circ \mathsf{NA}) \circ C_i = ?$", and replace t itself by $(T_2, (\mathsf{Pf} \circ \mathsf{NA}), T_1, C_i)$.

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

Computation of S in the initialization phase of GenARM encodes a total order that ensures RETs are free of cycles. RETs not free of cycles might happen otherwise, e.g., where two RM tables had complementary foreign keys to each other from their primary keys. Regarding the conversion to an ARM schema, note the replacement of foreign key constraints with unary counterparts or with specialization constraints (lines 5.4.1, 5.4.2), the addition of disjointness constraints (line 5.3), path functional dependencies (line 6). Both are necessary to ensure RET is ultimately identity resolving in the sense outlined above.

The addition of disjointness constraints between any pair of tables with primary keys that differ in any way are justified by virtue of the fact that an input RM schema has no way of detecting when respective primary key values could co-refer otherwise. For example, if students can also be professors, then it becomes impossible to compile a SQLP query for *department names of professors who are not students*.

The GenARM procedure achieves the first of our main goals:

Proposition 8 Let Σ_1 be a RM schema and Σ_2 and R the ARM schema and referring type assignment generated by $\textsc{GenARM}(\Sigma_1)$, respectively. Then for every SQLP query φ and database instance DB over Σ_2 we have

$$\textsc{SQLPtoSQL}(\varphi)(\textsc{Concrete}(DB, R)) = \textsc{Concrete}(\varphi(DB))$$

where the functions Concrete and SQLPtoSQL map ARM instances and SQLP queries to their corresponding relational counterparts using the referring type assignment R.

Note that, for the above proposition to hold, it is essential that the referring type assignment R produced in Step 6 of GenARM is *identity resolving* in order to map equalities between `OID` attributes in Σ_2 to equalities over attributes in Σ_1. Case analysis of the procedure shows that indeed R is identity resolving.

4 User Evaluation

The aim of the user evaluation is to ascertain whether querying with the ARM and SQLP has advantages over querying with the RM and SQL. We expect that it will take less time to construct the SQLP queries and they may also have fewer errors, for one does not have to painstakingly declare all the joins individually (recall also Example 4), which reduces the cognitive load as well as the size of the query, which may affect comprehension and authoring of queries. Two variables measure the potential difference: time taken and correctness, which lead to the following null (H) and alternative (A) statistical hypotheses:
H_t : there is no difference between SQL and SQLP in the mean time taken;

A_t : reading and writing in SQLP is faster than in SQL in the mean time taken;
H_c : there no difference between SQL and SQLP in the mean correctness;
A_c : SQLP queries have a higher level of correctness than SQL queries.
Because we assume SQLP will show an advantage over SQL, they will be assessed against the one sided alternatives, rather than the weaker two-sided option.

4.1 Experimental Design

Methods. Participants were recruited from undergraduate Computer Science major students at the University of Waterloo (UW)'s third year database class (CS348) and graduate students in Computer Science from UW's Data Systems Lab. (The experimental design was reviewed and approved by the Human Research Ethics Committee of UW before recruitment of subjects.) The set-up of the experiment uses a cross-over design for the graduate students, and a simple comparison for the undergraduate students. Undergraduate students were treated differently, because only one-third of them turned out to have had SQL experience, whereas all graduate students had. Half of the undergraduates were randomly assigned to answer the six questions using SQLP (group U_p, $n = 5$) and the other half to answer the SQL questions (group U_s, $n = 4$). The graduate students answered all six questions using the cross-cover approach; they were randomly assigned to answer either first the SQL questions and then SQLP (group G_s, $n = 8$) and the other in reverse order (group G_p, $n = 7$). Of the undergraduates, only 2 were native English speakers (both had been randomly assigned to U_p); of the graduate students, 5 were native English speakers (4 of these had been randomly assigned to G_p).

The experimental protocol is as follows. All subjects were given five minutes to read instructions about the protocol and then 10 minutes to read instructions and examples on the use of SQL or SQLP, or both in case of the cross-over design. The subjects were not allowed to ask questions. Subsequently, each subject received each question sealed in its own envelope, noting that no student knew which query language would be required until their session began. Subjects recorded (to the nearest second) their start time on an envelope when they opened it as well as their completion time when their answer to that question was returned in the envelope. Questions Q1-Q6 (see below) were answered by all subjects in that order and no previous question could be consulted. The questions in a question set used only one of SQL or SQLP. All answers were given in hand-written form and subjects had no access to any other electronic device or information source. There was no time limit. Given the international nature of our students, the subjects also answered the question whether their first language was English or not. Upon completion of the experiment, each subject received a gift voucher.

Performance is measured on the time taken to complete each question and the correctness of each answer. Time taken is based on the self-reported time (see procedure). To measure correctness, answers for each question are independently scored by three assessors (authors WM, DT, and GW) and the scores averaged; the assessors are blinded to all aspects about the subject and the experimental

conditions. The assessors agreed to score on a 4 point scale with half points allowed, where 0 meant to them completely wrong; 1: meant 'does not solve the question but the subject has grasped the basic concepts' (of SQL or SQLP); 2: meant 'the answer contains mistakes, but is on the right track and joined most of the required tables correctly'; 3: meant 'mostly correct, may only contain minor mistakes'; and 4: meant 'solves the question completely and correctly'.

Regarding the statistical analysis, different methods can be applied depending upon whether the results of all students are combined, or separated by graduate/undergraduate. We will analyze both, as both have advantages (more data points and the cross-over insights, respectively).

Materials Six questions were devised to cover both query comprehension and authoring: Q1-Q3 present code in SQL or SQLP and subjects have to provide a written summary of the query in English (comprehension); Q4-Q6 present a query in English and subjects have to code the query in SQL or SQLP (authoring), alike depicted in Example 4. The comprehension and authoring tasks focus on conjunctive cases involving no more than six table variables, except for Q2 that includes a not exists predicate. Authoring tasks were designed to be progressively more difficult (requiring more joins), as were comprehension questions Q1 and Q3. Figs. 1 and 2, without marking abstract attributes with '*', are the RM and ARM schemata used in the experiment. Details can be found in [8].

4.2 Results and Discussion

We will first present the combined results, and then the cross-over results.

Performance: all subjects combined. There are a total of 39 answers (20 for SQLP, 19 for SQL) for each performance measure with $30 = 2 \times 15$ from the graduate students ($|G_p| + |G_s|$) and $9 = |U_p| + |U_s|$ from the undergraduates. The fact that each graduate student provides values for both SQLP and SQL is of little concern since the experimental design randomly assigned half of them to using SQLP first and half to using SQL first, thus balancing any order effect. Fig. 4 shows the estimated values and 95% confidence intervals for the expected (or mean) performance for the correctness (left), and time taken (right). In both plots, SQL results are solid colored red, SQLP dashed blue.

Consider only the results for the comprehension questions Q1-Q3 for correctness. Looking at the SQL curve, we see increasing difficulty of these questions (as by design) in the decreasing mean values and increasing variability (interval length). The same pattern holds when performance on Q1-Q3 is measured by the expected time taken, though Q2 and Q3 are essentially indistinguishable.

Similarly, following the SQL curve for the authoring tasks given by Q4-Q6, we see increasing difficulty as measured by correctness and by time taken. For correctness, the difficulty shows as decreasing mean performance; for time taken, it shows dramatically as increasing mean time taken and as increased variability. A comparison of the SQL curves in both plots shows students taking less time

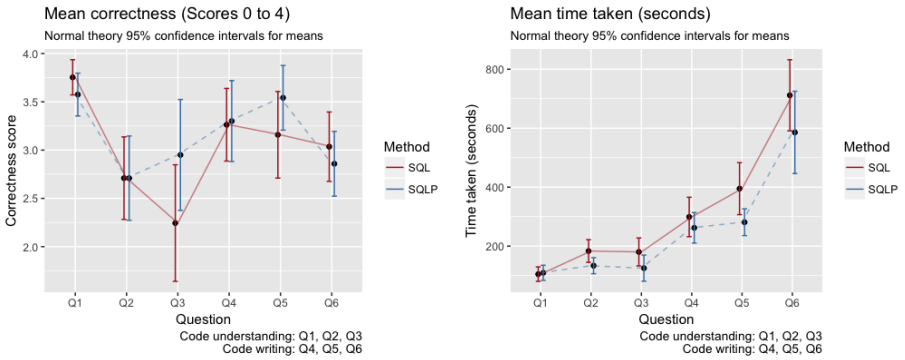

Fig. 4. Mean performance for all subjects: SQL solid; SQLP dashed.

and scoring more poorly on code reading (comprehension) questions and much more time but generally better scores on the code writing (authoring) questions.

More interesting are the results for the SQLP curve: except for Q1, *the expected time to complete each question using SQLP is consistently lower, and typically has smaller variability than, when using SQL!*

These estimates and intervals were based on standard normal theory (t-based confidence intervals), so to be conservative we also performed non-parametric two-sample Wilcoxon (or Mann-Whitney) tests comparing the sample times taken for SQL to those for SQLP for each question. The one sided test (H_t versus A_t) was performed giving the following p-values for each question: Q1 ($p = 0.63$), Q2 ($p = 0.021$), Q3 ($p = 0.018$), Q4 ($p = 0.27$), Q5 ($p = 0.009$), and Q6 ($p = 0.03$). Each p-value is the probability (assuming SQL and SQLP have the same distribution) of observing at least as large a difference as we did observe as measured by this test. The smaller the p-value is, the stronger the evidence against the null hypothesis (H_t) and in favor of the alternative (A_t). All but Q1 and Q4 would be judged to be highly statistically significant; *SQLP outperforms SQL in time taken.*

In contrast, *no statistically significant difference between SQLP and SQL in correctness was found* when the same tests were applied to the correctness scores. For correctness, the test yielded p-values for each question of Q1 ($p = 0.90$), Q2 ($p = 0.63$), Q3 ($p = 0.097$), Q4 ($p = 0.41$), Q5 ($p = 0.07$), and Q6 ($p = 0.77$). Such high probabilities mean that, as measured by this test, the data were consistent with the null hypothesis (H_c). If anything, in the two cases (Q3 and Q5) approaching statistical significance, the (left hand) plot shows SQLP outperforming SQL in correctness.

Finally, we consider how the performance fares with respect to English as first language or not. Fig. 5 shows the confidence intervals for the mean time taken in these cases. Again, the expected time taken using SQLP is often the same or lower than when using SQL and often with less variability. Most striking are those cases where the student's first language was not English; except for Q1, they appear to perform more quickly using SQLP than they do using SQL.

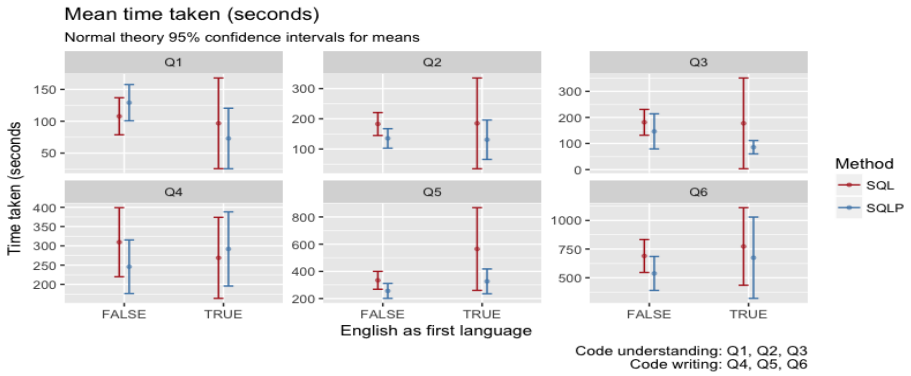

Fig. 5. Mean time taken estimates and confidence intervals by question separating the results of subjects based on whether their first language was English or not.

Performance: Graduate students only. The randomized cross-over design allows us to work with the difference between performance results for each student on each question and performance measure. The differencing should reduce variation between students. Fig. 6 shows the differences plotted by question for each performance measure. As before, with the possible exception of Q5 ($p = 0.00813$ for a one-sided test) where SQLP clearly outperforms SQL, *there is no statistically significant difference between SQLP and SQL observed for correctness* of the answers. Also as before, with the exception of Q1 *SQLP significantly outperforms SQL for both comprehension and authoring in time taken* with p-values (one-sided alternatives) by question: Q1 ($p = 0.500$), Q2 ($p = 0.00269$), Q3 ($p = 0.00488$), Q4 ($p = 0.0661$), Q5 ($p = 0.00413$), and Q6 ($p = 0.0820$). The last p-value is affected by the single outlying student near -1000 (Q6); as the plot shows, the remaining points for Q6 would have produced a significant value.

Because graduate students used both SQL and SQLP on the same questions, we can investigate the effect on performance of the order in which the methods were used. A priori, we expect that performance will improve when students faced the same questions again, albeit via a different model (RM+SQL or ARM+SQLP). The order effect was as expected for correctness, with the exception of Q5 where SQLP produced better answers than did SQL independently of the order. The results for time taken are shown in Fig. 7: as expected, when SQL is used first, students took longer to answer than when they subsequently used SQLP (points being above the horizontal line in the left hand plot) on the same questions. In contrast, when students used SQLP first, again with Q1 as an exception, their performance with SQL was surprisingly *not* a marked improvement on SQLP (points were *not* below the horizontal line in the right hand plot). That is, SQL still took longer than SQLP even though it was used *after* SQLP on the *same* questions.

Discussion of the results of the experiment The results of the experiments demonstrated that noted theoretical advantages of SQLP over SQL translated to SQLP outperforming SQL in time to completion (being consistently lower and having

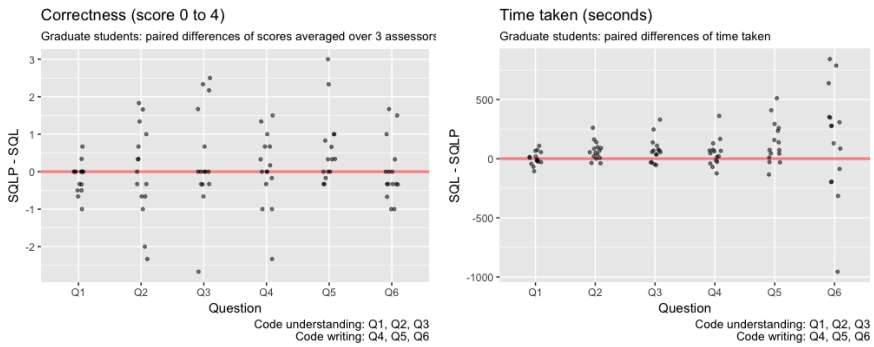

Fig. 6. Performance differences. Values above the horizontal line favor SQLP over SQL in both plots; random horizontal jittering separate points by question.

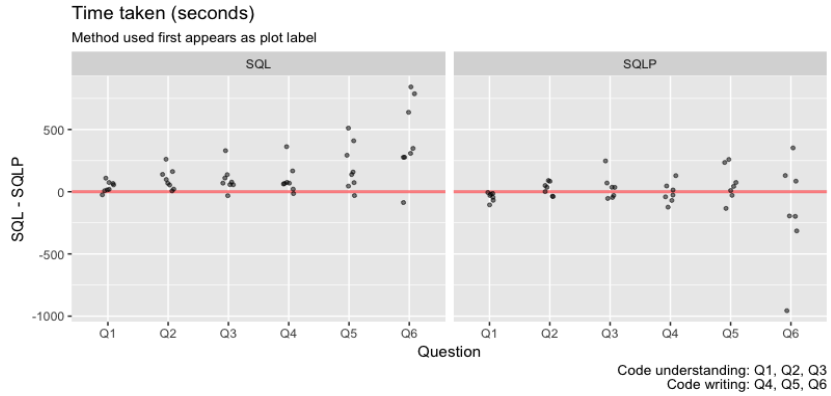

Fig. 7. Order effect: SQL took longer than SQLP when SQL is used first (left); With the possible exception of Q1, SQL takes about the same length of time when SQLP is used first (right). Random horizontal jittering used to separate points for each question.

less variation), i.e., A_t was accepted. H_c could not be rejected, although wherever observed differences approached statistical significance, they also favoured SQLP over SQL. Also, the learning curve for SQLP appeared to be low, given that no participant knew about SQLP or ARM before they began the experiment and were given only 10 minutes to learn about it.

There was no statistically significant difference in performance for Q1. It was the easiest question, and as difficulty increased, the differences between SQLP and SQL often increased (recall Fig. 4, right). Therefore, examining in further detail the effects of query difficulty levels and different types of queries is a promising direction for future experimental work, as it may refine insights into the practical advantages of using SQLP over SQL.

In other experiments that vary the model (or notation thereof) or the query language, a difference in either semantic or syntactic accuracy and in time take is observed when notation is the variable [3, 6], or both when the query language

abstracts away from plain SQL [7], and both when both are variables [10]. Our results on different query languages show a similar trend. The look-and-feel of the ARM diagram was made to look alike an RM one, in order to minimize the possibility that any difference observed could be attributable to the representation of the information rather than the query language. A future HCI experiment may be to devise more notations for ARM that have more or less vertical partitioning so as to examine those effects, which may then benefit also the accuracy.

5 Related work

While many path query languages have been proposed, to the best of our knowledge, there has been only one experimental evaluation to compare it to a 'non-path' version. Junkkari and co-authors used PathSQL, which in their experiment showed that it reduces query writing time and have fewer errors cf. SQL [7][4]. PathSQL [12] constructs paths for queries over aggregation hierarchies as a way to represent a series of left outer joins more compactly. In contrast, SQLP's paths can be constructed over joins in the direction of the functional dependencies.

Due to lack of other related work, we broadened the scope on conceptual queries that still relate to relational models. It has been shown that models at a higher level of abstraction have either equal or higher accuracy (fewer semantic errors) in the queries and are always formulated in less time when querying with the aid of a relational or conceptual model [6, 10]. Fewer syntax errors were observed when the aid was a textual relational model, but it was slower cf. a graphical depiction of the relational model, with no difference in accuracy of the queries [6]. It is not clear where the border lies between how detailed the graphically depicted relational model has to be to be optimal, but some parsimony seems to be favored especially for more complex SQL queries [3]. Thus, the results obtained in our experiment is in line with related, albeit different, experiments. Examining Bowen et al's parsimonious and detailed diagrams [3], the ARM depicted in Fig. 2 may strike a good balance, but this deserves further assessment by HCI experts.

Querying at the conceptual layer recently has gained interest because of the relative popularity of Ontology-Based Data Access (OBDA). OBDA typically supports only a subset of SQL, however, whereas SQLP supports full SQL plus path queries and one does not have to learn a new query language (SPARQL). While the ARM in Fig. 2 does not look like an ontology, it is just a graphical display at a certain partitioning level to make it look like an RM for the experiment. Each such rectangle there can be partitioned into unaries and binaries thanks to lossless vertical partitioning (Section 2), so then together with ARM's formalisation in the DL $\mathcal{CFDI}_{nc}^{\forall-}$ [11] that can fully simulate DL-Lite$_{core}^{\mathcal{F}}$ (a member in the DL-lite family used for OBDA with OWL 2 QL) [5], one easily can transform such an ARM into the same kind of ontology as used in OBDA.

[4] as stated in their abstract; despite efforts, we were not able to obtain the full paper.

6 Conclusions

Querying for information with the SQLP path query language and the Abstract Relational Model has been shown to be significantly faster than the baseline of SQL with the Relational Model, whilst maintaining accuracy. Thanks to the referring expression types and the lossless vertical partitioning it permits, an Abstract Relational Model can be made look like either a conceptual data model or function as a relational model. Therewith it can take advantage of both the benefit of conceptual queries and the full SQL support with the relational model within one formalism, whilst keeping an actual SQLP queries invariant in the face of such partitioning decisions. We have proposed a novel method to reverse engineer legacy relational models up to abstract relational models to facilitate its uptake, which, thanks to the automated analysis of the keys, also uncovers implicit constraints in the model, such as subsumption and disjointness.

References

1. Borgida, A., Toman, D., Weddell, G.: On referring expressions in query answering over first order knowledge bases. In: Proc. of KR'16. pp. 319–328. ACM (2016)
2. Borgida, A., Toman, D., Weddell, G.E.: On referring expressions in information systems derived from conceptual modelling. In: Proc. of ER'16. LNCS, vol. 9974, pp. 183–197. Springer (2016)
3. Bowen, P.L., O'Farrell, R.A., Rohde, F.H.: An empirical investigation of end-user query development: The effects of improved model expressiveness vs. complexity. Info. Sys. Res. 20(4), 565–584 (2009)
4. Calvanese, D., Keet, C.M., Nutt, W., Rodríguez-Muro, M., Stefanoni, G.: Web-based graphical querying of databases through an ontology: the WONDER system. In: Proc. of ACM SAC'10. pp. 1389–1396. ACM (2010)
5. Jacques, J.S., Toman, D., Weddell, G.E.: Object-relational queries over $\mathcal{CFDI}_{nc}$ knowledge bases: OBDA for the SQL-Literate. In: IJCAI'16. pp. 1258–1264 (2016)
6. Jih, W.J., Bradbard, D.A., Snyder, C.A., Thompson, N.G.A.: The effects of relational and entity-relationship data models on query performance of end-users. Int J Man-Machine Stu 31(3), 257–267 (1989)
7. Junkkari, M., Vainio, J., Iltanenan, K., Arvola, P., Kari, H., , Kekäläinen, J.: Path expressions in SQL: A user study on query formulation. J of DB Mgmt 22(3), 22p (2016)
8. Ma, W.: On the Utility of Adding an Abstract Domain and Attribute Paths to SQL. Master's thesis, University of Waterloo (2018)
9. Mylopoulos, J., Bernstein, P.A., Wong, H.K.T.: A language facility for designing database-intensive applications. ACM TODS 5(2), 185–207 (1980)
10. Siau, K.L., Chan, H.C., Wei, K.K.: Effects of query complexity and learning on novice user query performance with conceptual and logical database interfaces. IEEE Trans. Sys., Man Cybern. 34(2), 276–281 (2004)
11. Toman, D., Weddell, G.E.: On adding inverse features to the description logic $\mathcal{CFD}^{\forall}_{nc}$. In: Proc. of PRICAI'14. pp. 587–599 (2014)
12. Vainio, J., Junkkari, M.: SQL-based semantics for path expressions over hierarchical data in relational databases. J Info Sci 40(3), 293–312 (2014)
13. Weddell, G.: Reasoning about Functional Dependencies Generalized for Semantic Data Models. TODS 17(1), 32–64 (1992)