Induction of Relational Algebra Expressions

Joris J.M. Gillis and Jan Van den Bussche

Universiteit Hasselt and transnationale Universiteit Limburg

1 Introduction

In the theory of database systems [1], a *database query* is defined as a function that maps relational databases to relations. This definition models the situation in practice where one applies an SQL query to a database instance and receives a set of output tuples as the answer to the query on that database. The problem of *relational database query induction* is then naturally stated as follows: we are presented with a finite number of examples, where each example consists of a relational database and an output relation, and we are asked to come up with an expression for a query that agrees with the given examples (and that satisfies the usual requirements placed on induction tasks, most notably generalization).

Of course, we should also specify here which language we are using to express database queries. Whereas SQL is the universal query language used in practice, it is also very complex. A more tractable language to work with is the *relational algebra*, which can be found, together with SQL, in every database textbook, as the relational algebra is the basic underlying query language used for database query processing.

Another common language in which relational database queries can be expressed is Datalog (function-free Prolog). Datalog is also used in Inductive Logic Programming (ILP), and when we restrict attention to nonrecursive programs, Datalog can be translated into the relational algebra [2, 3]. As a consequence, relational database query induction could be (and has been [4, 5]) considered to be a mere special case of ILP, where the background knowledge consists only of facts (the example databases).

That approach, however, does not cover relational database query induction in its entirety, because not every relational algebra query can be expressed in Datalog. Nonrecursive Datalog without negation can only express the *positive* fragment of the relational algebra, i.e., the fragment without the set-theoretic difference operator. Consider, for example, the following rules over a binary relation $r(A, B)$:

$q(A, B) \leftarrow r(A, C),\, r(C, B)$
$q(A, B) \leftarrow r(A, C),\, r(C, D),\, r(D, B)$

The query q corresponds to the following positive relational algebra expression:

$$\pi_{A,B}(\rho_{B/C}(r) \bowtie \rho_{A/C}(r)) \cup \pi_{A,B}(\rho_{B/C}(r) \bowtie \rho_{\substack{A/C\\B/D}}(r) \bowtie \rho_{A/C}(r))$$

Of course, one might use Datalog with negation, and then the difference between two relations r and s can be expressed as follows:

$diff(A, B) \leftarrow r(A, B), \neg s(A, B)$

If negation can only be placed before input relations, however, this does not suffice to express all relational algebra queries. Notably queries involving universal quantification can still not be expressed. A classical example of such a query is Codd's *relational division* [6] of two relations $r(A, B)$ and $s(B)$, which returns the following unary relation as answer:[1]

$$r \div s = \{A \mid \exists B : r(A, B) \land \forall B : (s(B) \rightarrow r(A, B)\}$$

In relational algebra, division is expressible as $\pi_A(r) \setminus \pi_A((\pi_A(r) \times s) \setminus r)$. Note the nested application of the difference operator $\setminus$. If we want to express such queries in Datalog with negation, we need to introduce auxiliary predicates. In general it is known that with deeper and deeper nested applications of the difference operator (corresponding to alternations of existential and universal quantifiers), more and more queries can be expressed [7], so there is no bound on the number of required auxiliary predicates.

We conclude that Datalog-based approaches to induction of relational algebra queries require a combination of predicate invention and negation. Predicate invention received quite a bit of attention in ILP until the mid 1990s (e.g., [8, 9]), and is recently receiving renewed attention [10]. Relatively few ILP systems support negation (e.g., FOIL [11] and TILDE [12]). It is quite conceivable that by combining various of these earlier ILP techniques, relational algebra queries can be induced successfully. In the present paper, however, we explore an approach that directly searches for relational algebra expressions. Nevertheless, we hope that some of our ideas can also be of use in a more classical ILP-based approach. For example, because relational algebra expressions can be translated back into nonrecursive Datalog programs with stratified negation [1], our approach could also be viewed as a form of predicate invention.

Related work. We end this introduction with a brief review of some related work, apart from the large body of work in ILP [13]. Acar and Motro [14] focus on the induction of selection queries only (queries without joins) and thus their work essentially boils down to attribute–value learning. In the Clio system [15], mappings between database schemas are inferred in the form of database queries, but the examples given to the induction algorithm are much more fine-grained and consist of explicit value paths between data elements from the source database and the target database. Closest in spirit to our own work is Tupelo [16], a system for inducing data mappings. Tupelo focuses on data restructurings, which are lossless and therefore easier to induce, and is based on a special-purpose algebra with specific restructuring operators (the algebra also lacks the difference operator). Nevertheless, we were influenced by the overall design of Tupelo's search algorithm, which we have copied in the present work.

[1] If r would contain patients with observed symptoms, and s would contain the set of required symptoms to qualify for some specific disease, then $r \div s$ would return all patients that can be diagnosed with the disease.

2 Relational and cylindric set algebra

We assume some familiarity with relational databases and relational algebra, but we fix some terminology and notations.

A relational *database schema* is a finite set of relation names, each with a *relation scheme*, which is a finite set of attribute names. If relation name R has scheme $\{A, B, C\}$, this is often denoted as $R(A, B, C)$. The relational algebra consists of the six operators set union $\cup$; set difference $\setminus$; natural join $\bowtie$; selection $\sigma_{A=B}$; projection π_Z; and renaming $\rho_{A/B}$. Here, A and B stand for attribute names, and Z for a finite set of those. Expressions, of the kind illustrated in the Introduction, are built from relation names using these operators. Each expression has a result relation scheme that can be syntactically determined from the input relation schemes. Semantically, a relational algebra expression over some database schema is evaluated on a database of that schema; the result relation of expression e evaluated on database D is denoted by $e(D)$.

The relational algebra is a typed language: an operator can be applied to expressions only if their result schemes are compatible with the operator. For example, union $e_1 \cup e_2$ can be formed only if e_1 and e_2 have the same result scheme, and renaming $\rho_{A/B}(e)$ can be formed only if the result scheme of e contains attribute A but not attribute B. All these typing restrictions can be something of a nuisance in an induction setting, where we want to systematically combine and generate expressions. Moreover, the subexpressions of an expression can have different result schemes, that can be also different from the final result scheme. In a bottom-up approach where we build up more and more complex expressions, this implies that when testing a hypothesis, we must compare relations of different schemes. Indeed, the given examples are of the final result scheme, whereas intermediate expressions to be tested produce relations of a different scheme.

To avoid these problems, we propose the use of the *cylindric set algebra (CSA)* [17–19] as a more suitable alternative to the relational algebra in the context of induction. Fix some finite set U of attribute names. The operators of CSA over U deal only with relations of scheme U. There are only four operators: union; complementation; selection $\sigma_{A=B}$; and *cylindrification* γ_A, with $A, B \in U$. Union and selection are as in the standard relational algebra. Complementation is the classical set-theoretic operator, but, in order to guarantee finiteness, relativized to the *active domain* $adom(D)$ of the database D on which we are evaluating the expression. The active domain is the set of all values appearing in the relations of D. Then the complement of a relation over U, relative to D, is the set of all tuples over U that take values in $adom(D)$ and that do not belong to r. Finally, the cylindrification $\gamma_A(r)$ of a relation r over U is the set of all tuples over U that agree with some tuple in r on $U \setminus \{A\}$. This intuitively corresponds to existential quantification on the A-column.

As in the relational algebra, given some database schema where all relation names now have scheme U; we call such a schema U we can build up CSA expressions from relation names using the four CSA operators. Unlike the standard relational algebra, there are no restrictions on the formation of expressions, be-

cause everything is of the same type U. Formally, let a database schema be given that is U*-uniform* in the sense that every relation name has scheme U. Then each relation name is an expression, and if e_1 and e_2 are expressions, then so are $(e_1 \cup e_2)$, $(e_1)^{\mathrm{c}}$, $\sigma_{A=B}(e)$, and $\gamma_A(e)$ without any restrictions. The result relation of any expression is a relation of scheme U. This makes the CSA very flexible to work with in an induction context.

It can be proven [18] that we do not lose any expressive power by working with CSA instead of the standard relational algebra, as long as we work only with the attributes from U. Difference is expressed using complementation and intersection (a non-primitive operator that we add for convenience); join becomes plain intersection; projection becomes cylindrification; and renaming is simulated using cylindrification and selection.

The choice of U is of course an important parameter. Clearly, we will put in U all attributes from the input database schema and from the output relation scheme. But additional attributes may be needed to be able to express the query. For example, over a binary relation $R(A, B)$ interpreted as the set of edges of a directed graph, the query "output all nodes where a path of length at least two originates" is expressible over $U = \{A, B, C\}$ but not over $U = \{A, B\}$.

3 Heuristic values taking into account complement

For the induction of an unknown relational database query Q, we are presented with one or more examples of the form (D, r), where D is a database and r is a relation, and the assumption is that $r = Q(D)$. During induction we are searching for an expression e for Q. Thereto we will test hypothesis expressions against the examples to see how well they agree. Thanks to uniformization and the use of CSA, as explained above, both the relation $e(D)$ and the relation $r = Q(D)$ have the same scheme U, which makes comparison quite standard. Indeed, a commonly used metric $d(X, Y)$ on subsets X and Y of some finite universe T uses the symmetric difference: $d(X, Y) = (|X \setminus Y| + |Y \setminus X|)/|V|$. In our case, X and Y are relations of scheme U taking values in the active domain of D, so $|V| = |adom(D)|^{|U|}$.

This simple approach is insufficient, however, for queries whose expression requires set difference (or in CSA, complementation). Indeed, such expressions typically compute a relation that equals the complement of the desired output relation, then perform a final complementation. Consider, for example, the basic *universal quantification* query about two database relations $R(A, B)$ and $S(B)$ that outputs the relation $\{A \mid \exists B : R(A, B) \land \forall B : (R(A, B) \rightarrow S(B))\}$ (this query is similar to relational division but the implication is in the other direction). In CSA it is expressible as $\gamma_B\left(R \cap (R \cap S^{\mathrm{c}})^{\mathrm{c}}\right)$. The subexpression $R \cap S^{\mathrm{c}}$ is crucial, but a discovery algorithm based only on direct comparison with the example outputs would discourage further elaboration of this subexpression, as it results in a relation that is extremely different from the final desired result.

We settled on a very direct and simple solution: in order to encourage the exploration of expressions involving complementation, as the heuristic value for

an expression e on an example (D, r), we do not use $d(e(D), r)$ but rather $\min\{d(e(D), r), d(e(D)^{c}, r)\}$. In this way, hypotheses that come close to the example, as well as hypotheses that come close the complement, are favored.

4 Searching for expressions

As in Tupelo [16], our search space consists of *straight-line programs:* finite sequences of statements. Here a *statement* has the form $R := \mathrm{op}(R_1)$ or $R := R_1 \mathrm{\ op\ } R_2$, where op is a CSA operator, R is a relation variable, and R_1 and R_2 are either relation names from the database schema or relation variables introduced earlier in the program. Within a program, each relation variable represents an expression, obtained by tracing out its definition. For example, in the program $R_1 := S^{c}$; $R_2 := \gamma_A(R)$; $R_3 := R_1 \cup R_2$, variable R_3 stands for the expression $S^{c} \cup \gamma_A(R)$. Important for us are the *top-level* variables in a program, that are those variables that do not occur on the right-hand side of a statement in the program. There can be several top-level variables in a program, and we view a program as representing a set of expressions, one for each top-level variable. Accordingly, as the heuristic value of a program, we use the average heuristic values of its top-level expressions.

The search space can be kept finite because, for each example database D (of which there are a finite number), there are only a finite number of relations of scheme U taking values in $adom(D)$. When exploring the search space by adding a statement to some already encountered program, if that statement defines the same relation[2] on all example databases as some previous statement in the program, the statement will not be added to the program. By imposing an order on the operators and respecting the alphabetical order on attributes in the equivalence operator, we can avoid generating equivalent programs (on the given set of example databases) more than once. If an operator δ uses a variable X as an operand, X becomes blocked for operators that come before δ in the order. It is easy to show that this restriction does not diminish the expressivity of the search space. We note that, even if U consists of just three attributes, full equivalence of CSA expressions (on all possible databases rather than just the given examples) is undecidable [20]. One can easily imagine other optimizations to trim down the size of the search space. It is, for example unwise to generate expressions containing: $\sigma_{A=B}(\sigma_{A=B}(\ldots))$. The outermost selection operator has no effect whatsoever.

Everything is now in place to find, if it exists, an expression for the unknown query Q given by example. If an U-bounded expression exists that agrees with all examples, it will be found. We perform a best-first search [21] using the heuristic on programs explained above. The initial state is the empty program, which is understood to have as top-level expressions the relation names from the database schema. We generate successor states by appending an extra statement to a program.

[2] In practice, such equivalence checks can be implemented efficiently using hashing, at the price of, at least in principle, possible false positives.

5 Experimental results

We have tested our ideas on a number of typical relational database queries involving universal quantification. We have also tried to induce the queries using the ILP systems FOIL [11] and TILDE [12].[3]

The two obvious testcases were *Codd's relational division* (discussed in the Introduction) and the *universal quantification* query (discussed in Section 3), both about relations $R(A, B)$ and $S(B)$. Both are expressible in CSA over $U = \{A, B\}$, i.e., no additional attribute names are needed, using expressions we have seen earlier. Expressions for these queries were inferred by our search algorithm in a mere 17 (for division) respectively 5 (for universal quantification) steps (where a step means the expansion of a state to all its successor states). Interestingly, whereas for division the obvious expression was found, for universal quantification an equivalent, rather ingenious expression was found that involves more complementation steps but that is shorter overall, namely, $(\gamma_B(R \cup S^{\mathrm{c}})^{\mathrm{c}})^{\mathrm{c}}$.

It must be said, however, that representative examples are important. For division, we presented the algorithm with a small number of databases generated from a small number of constants a_i and b_j which will serve as the nodes of the directed graph R. The b's go in relation S. We generate edges randomly but make make sure to have some nodes with edges to all b's (and also some a's); these nodes will form the output part of the example. We make sure the graph looks sufficiently random otherwise, with edges from a's to a's, from a's to b's, from b's to a's, and from b's to b's. Figure 1 shows a small example database. For universal quantification, we did something similar.

Next we provided FOIL and TILDE with the same examples, to induce both example queries. Both FOIL and TILDE are able to produce Datalog rules with negation. Thus easier queries involving negation, such as the plain difference between two relations, are no problem for both systems. FOIL however was not able to induce the correct expression for the queries with universal quantification. The universal quantification and relational division query are expressible as *First-Order Logical Decision Trees* (FOLDTs). The corresponding trees are shown in table 1. Tilde induced both trees correctly. Again we have to note that representative examples are very important for the induction process.

Obviously, as always, there is no free lunch, and there are queries that take a long time to find using our simple search-based approach. Especially queries that require the renaming operator for their expression seem difficult to find. Renaming is an example of a lossless data transformation of the kind targeted by Tupelo [16]. It might be worthwhile to try to use Tupelo search inside the determination of the heuristic value of a candidate expression (search within search), so that pure data transformations are evaluated separately from query operators. This deserves further investigation.

---

[3] For TILDE we used the KULeuven ACE data mining system [22].

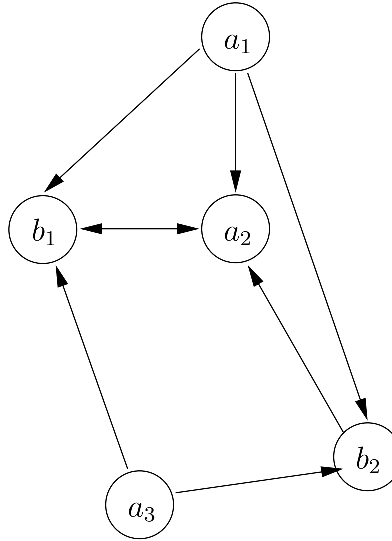

Fig. 1. A small example database for learning Codd's relational division. The database consists of three 'a'-nodes and two 'b'-nodes. Nodes a_1 and a_3 form the output of the relational division.

Table 1. First-Order Logical Decision Trees expressing the universal quantification query and the relational division query.

Universal Quantification

```
q(-A,-B,-C)
r(A,B,-D) ?
+--yes: not_b(A,D) ?
| +--yes: [neg]
| +--no: [pos]
+--no: [pos]
```

Relational Division

```
q(-A,-B,-C)
s(A,-D) ?
+--yes: not_r(A, B, D) ?
| +--yes: [neg]
| +--no: [pos]
+--no: [pos]
```

6 Conclusion

Our purpose in this paper was to draw attention to the induction of first-order database queries involving universal quantification, and to the potential of an algebraic approach to ILP. Clearly one can go back and forth between the relational algebra and Datalog, and hopefully some of the ideas we have offered can also be relevant in a more standard ILP setting. Note also that our restriction to relations over a fixed scheme U could be explained as a rudimentary language bias mechanism.

An algebraic approach can sometimes be more convenient. Think, for example, about a genetic programming approach [23] to learning relational algebra expressions. The CSA is convenient in this respect as genetic programming operations on expressions such as mutation and crossover can be freely performed, always resulting in well-formed expressions that can be effectively evaluated.

Acknowledgment We thank Jan Struyf and Hendrik Blockeel for providing us with the ACE software package and teaching us how to use Tilde.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Ullman, J.: Principles of Database and Knowledge-Base Systems. Volume I. Computer Science Press (1988)
3. Ullman, J.: Principles of Database and Knowledge-Base Systems. Volume II. Computer Science Press (1989)
4. Blockeel, H., De Raedt, L.: Relational knowledge discovery in databases. In Muggleton, S., ed.: Inductive Logic Programming, Selected Papers 6th International Workshop. Volume 1314 of Lecture Notes in Computer Science., Springer (1997) 199–211
5. Blockeel, H., De Raedt, L.: Inductive database design. In Ras, Z., Michalewicz, M., eds.: Foundations of Intelligent Systems, Proceedings 9th ISMIS. Volume 1079 of Lecture Notes in Computer Science., Springer (1996) 376–385
6. Codd, E.: Relational completeness of data base sublanguages. In Rustin, R., ed.: Data Base Systems. Prentice-Hall (1972) 65–98
7. Chandra, A., Harel, D.: Structure and complexity of relational queries. Journal of Computer and System Sciences **25** (1982) 99–128
8. Silverstein, G., Pazzani, M.: Relational clichés: Constraining induction during relational learning. In Birnbaum, L., Collins, G., eds.: Proceedings 8th International Workshop on Machine Learning, Morgan Kaufmann (1991) 203–207
9. Kijsirikul, B., Numao, M., Shimura, M.: Discrimination-based constructive induction of logic programs. In: Proceedings 10th NCAI, AAAI Press (1992) 44–49
10. Kok, S., Domingos, P.: Statistical predicate invention. In Ghahramani, Z., ed.: Proceedings 24th International Conference on Machine Learning, ACM Press (2007) 433–440
11. Quinlan, J., Cameron-Jones, R.: Induction of logic programs: FOIL and related systems. New Generation Computing **13**(3–4) (1995) 287–312

12. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence **101**(1–2) (1998) 285–297
13. De Raedt, L.: Logical and Relational Learning. Springer (2008)
14. Acar, A., Motro, A.: Intensional encapsulations of database subsets via genetic programming. In Andersen, K., Debenham, J., Wagner, R., eds.: Database and Expert Systems Applications, Proceedings 16th DEXA. Volume 3588 of Lecture Notes in Computer Science., Springer (2005) 365–374
15. Miller, R., Haas, L., Hernández, M.: Schema mapping as query discovery. In: Proceedings 26th VLDB. (2000) 77–88
16. Fletcher, G., Wyss, C.: Data mapping as search. In Ioannidis, Y., et al., eds.: Advances in Database Technology—EDBT 2006. Volume 3896 of Lecture Notes in Computer Science., Springer (2006) 95–111
17. Henkin, L., Monk, J., Tarski, A.: Cylindric Algebras. Part I. North-Holland (1971)
18. Van den Bussche, J.: Applications of Alfred Tarski's ideas in database theory. In Fribourg, L., ed.: Computer Science Logic. Volume 2142 of Lecture Notes in Computer Science., Springer (2001)
19. Imielinski, T., Lipski, W.: The relational model of data and cylindric algebras. Journal of Computer and System Sciences **28** (1984) 80–102
20. Kahr, A., Moore, E., Wang, H.: Entscheidungsproblem reduced to the $\forall\exists\forall$ case. Proceedings of the National Academy of Sciences of the USA **48** (1962) 365–377
21. Russell, S., Norvig, P.: Artificial Intelligence, A Modern Approach. Second edn. Prentice Hall (2003)
22. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Raons, J., Vandecasteele, H.: Improving the efficiency of inductive logic programming through the use of query packs. Journal of Artificial Intelligence Researche **16** (2002) 135–166
23. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)