Journal of Information Systems Education

Volume 30
Issue 3
Summer 2019

Teaching Tip
A Notation for Planning SQL Queries

Toni Taipalus

Recommended Citation: Taipalus, T. (2019). Teaching Tip: A Notation for Planning SQL Queries. *Journal of Information Systems Education,* 30(3), 160-166.

Article Link: http://jise.org/Volume30/n3/JISEv30n3p160.html

Initial Submission:	April 4 2018
Accepted:	13 February 2019
Abstract Posted Online:	5 June 2019
Published:	12 September 2019

ISSN: 2574-3872 (Online) 1055-3096 (Print)

Teaching Tip
A Notation for Planning SQL Queries

Toni Taipalus
University of Jyvaskyla
Faculty of Information Technology
Jyvaskyla, Finland, 40014
toni.taipalus@jyu.fi

ABSTRACT

Structured Query Language (SQL) is still the de facto database query language widely used in industry and taught in almost all university level database courses. The role of SQL is further strengthened by the emergence of NewSQL systems which use SQL as their query language as well as some NoSQL systems, e.g., Cassandra and DynamoDB, which base their query languages on SQL. Even though the syntax of SQL is relatively simple when compared to programming languages, studies suggest that students struggle with simple concepts due to working memory constraints when learning SQL. This teaching tip presents a novel, simple, and intuitive notation for planning more complex SQL queries, which 1) facilitates the learning of SQL by providing students with a big picture of a particular data demand in regard to the database structure and 2) separates the logic of a data demand from the syntax and semantics of SQL, thus alleviating the strain on the student's short-term memory. The notation can also be applied when discussing SQL semantics during the teaching process without focusing on the syntactical nuances of the language.

Keywords: Structured query language (SQL), Query language, Data management, Data visualization, Teaching tip

1. INTRODUCTION

When teaching programming, teachers often emphasize planning before writing, and encourage the use of various techniques, e.g., flowcharts, to plan how the software works. As the software becomes increasingly complex, planning can be supported by design, e.g., by using class diagrams. Various planning techniques that support learning have been proposed for programming (e.g., Hu, Winikoff, and Cranefield, 2012), but SQL has received less attention despite its popularity in both education and industry. The techniques intended for supporting the learning of programming cannot be utilized as is with SQL because of the declarative (i.e., a query is a description of what) and set focused (i.e., a query is difficult or impossible to divide into working subsets) nature of SQL as opposed to the imperative (i.e., a function is a description of how) and step focused (i.e., software operates line-by-line and function-by-function) nature of programming languages such as Java, C#, or Python. These differences make the use of flowcharts unsuitable for planning SQL queries.

The more complex the query is, the more strain it puts on the query writer's short-term memory (e.g., de Jong, 2010, for working memory in general; Smelcel, 1995, for working memory in SQL in particular). Additionally, Ahadi et al. (2016) found that omission errors are among the most common errors when students are learning SQL and proposed that following a systematic procedure and segmenting the question could be the solution for avoiding omission errors. Additionally, even though the syntax of SQL is relatively simple, during the query writing process, the writer must recall SQL keywords with their syntax and semantics, in addition to the database object names, namespaces, and required expressions which, according to Smelcer (1995), often causes strain on the student's short-term memory. Furthermore, Buitendijk (1988) discussed that one of the four major reasons for writing incorrect SQL queries was the complexity of the task. Our work introduces a simple and intuitive notation for planning SQL queries (NPSQ) which is not based on any existing notation. The purpose of the notation is two-fold. First, to assist the student in acquiring the big picture of more complex queries, and second, to separate logic and semantics from syntax, thus alleviating the strain on the student's short-term memory.

The notation can be utilized in any database course that involves SQL. We have used the notation in an introductory database course with approximately 250 to 350 students (depending on the year), mandatory for undergraduate students who major in information systems or computer science, who typically have no previous experience in SQL. We have taught SQL from the SQL standard's perspective as proposed by Randolph (2003). In addition to positive student feedback, several industry professionals have indicated that the notation has proven increasingly useful when planning more and more complex queries.

2. BACKGROUND

In this section, we first define key terms for this work. We then describe our perceptions on how a query writing process takes

place in order to give background on what conceptions have driven the evolution of the notation.

2.1 Terminology

A *data demand* is a natural language representation of what data is needed to which a query writer, e.g., a student, is required to write an equivalent *query* in SQL. When a query is run, the database management system outputs an error message, or a *result table* which contains the rows that satisfy the query. A *query plan* is a picture drawn by a query writer using NPSQ. A query plan is drawn after reading the data demand but before writing the query.

Rows that satisfy a query can be limited in two ways: *joins* and *expressions*. To the extent of our teaching, a student can write a join in one of four methods: using the JOIN predicate, an uncorrelated subquery with IN, a correlated subquery with EXISTS, or with an explicit join condition without a subquery. Not all the methods can be applied for all data demands, and some methods fit more naturally to some data demands. Expressions concern either a column, or groups, which means that the expression is placed either in a WHERE clause, or a HAVING clause, respectively. Concrete examples of some of these methods can be found in Appendix 1, and examples of all methods in Taipalus, Siponen, and Vartiainen (2018).

2.2 The Query Writing Process

Over the last eight years of teaching SQL, we have identified six steps in the query writing process which, in turn, have guided the formulation and usage of this notation. Similar steps or aspects have also been recognized by others (e.g., Casterella and Vijayasarathy, 2013). These steps are, in order: i) which tables are needed to answer the data demand; ii) which columns are needed in the result table; iii) which tables need to be joined; iv) which columns are the joining columns, and is there a need for an outer join; v) which columns are subject to expressions; and vi) is there a need for ordering, grouping, or expressions on groups. These steps can be interpreted as one of the lower level presentations of the model of the query formulation process suggested by Borthick et al. (2001).

3. THE NOTATION

In this section we first discuss the elements of NPSQ from a more theoretical viewpoint and then present practical step-by-step instructions on how to utilize the notation. More examples can be found in Appendix 1. The notation can also be utilized for complex UPDATE and DELETE statements with little or no modifications. Furthermore, the notation may be used with other relationally complete query languages which, however, appear to be scarce.

3.1 The Elements of the Notation

NPSQ does not decree the syntactical elements of the query, e.g., which method should be used when writing joins or how expressions should be written, but only the logic of the query. We have designed the notation for SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING clauses, because these are the most commonly taught data retrieval elements of SQL. Although we present the elements of NPSQ drawn with a computer program, we emphasize that the planning should take place with pen and paper for quickness and convenience. Figure 1 summarizes the notation, and Table 1 presents examples of the SQL equivalents. The elements on the left side correspond to relational algebraic operations (Codd, 1970, 1972):

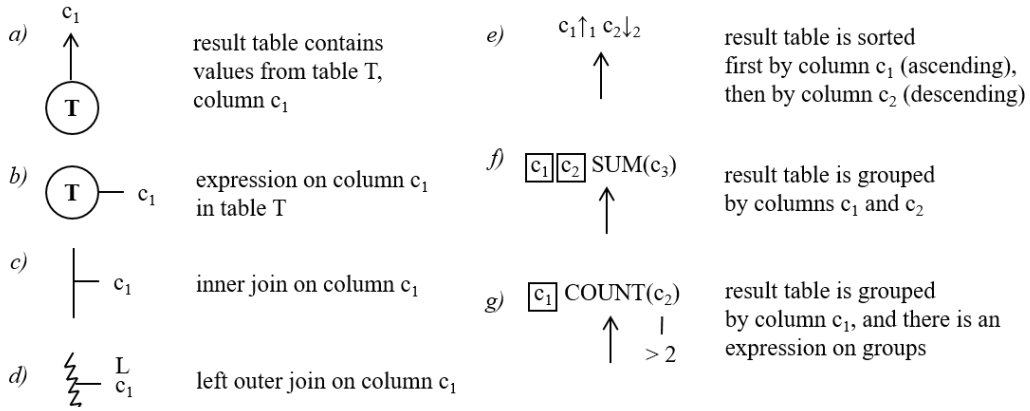


Figure 1. The Elements of the Notation

a) SELECT c_1 FROM <some table> T	*e)* ORDER BY c_1 ASC, c_2 DESC
b) FROM <some table> T WHERE c_1 = <some value>	*f)* SELECT c_1, c_2, SUM(c_3) […] GROUP BY c_1, c_2
c) FROM <some table> T INNER JOIN <some table> S ON (T.c_1 = S.c_1)	*g)* SELECT c_1, COUNT(c_2) […] GROUP BY c_1 HAVING COUNT(c_2) > 2
d) FROM <table> T LEFT OUTER JOIN <some table> S ON (T.c_1 = S.c_1)	

Table 1. The Corresponding SQL Concepts for Each Element of the Notation

projection (SELECT), restriction (WHERE), join (INNER JOIN), and intersection (OUTER JOIN). The elements on the right side correspond to SQL clauses: sorting (ORDER BY), grouping (GROUP BY), and expressions on groups (HAVING).

While we designed the elements of the notation around the six steps of the query writing process discussed in Section 2.2, the structure of a query plan is inspired by the query trees used as input and output by the query processing components of different database management systems. A query plan can also be understood as a graph with nodes (tables), edges (joins), and properties (joining columns and expressions) of both. Furthermore, a query plan is a kind of tree in which the root node is the table from which columns are projected into the result table. However, a tree can have multiple root nodes, if the result table contains columns from more than one table.

Tables should be represented not by table names but by short aliases for brevity and convenience. In a case such as a self-join when the same table must be presented more than once, different aliases should be considered, e.g., T_1 and T_2 for table *T*. If an expression is complex, or the expression repeated with different values for different tables, more precise notation can be used, e.g., c_1 = *'New York'* instead of c_1. If a join is complex, e.g., based on an aggregate function, or if a quantified comparison operator such as ALL is used, it can be presented as a property of the corresponding edge.

If the query is written with subqueries, the distance from the root node(s) represents the depth of a query; the root nodes represent the main SELECT clause, the nodes on the next level of the tree represent first level subqueries, the nodes on the level below that represent second level subqueries etc. A case of negated existential quantifier (¬∃) can be formulated with either left or right outer join, with a subquery using NOT IN or NOT EXISTS, or with ALL. In the former case, letters L or R can be used to illustrate the type of the outer join, as demonstrated in Figure 1 (d). If NATURAL JOIN or CROSS JOIN is used, the property of the edge can be omitted.

In the scope of our course, we teach only strict grouping. In practice, this means that if an aggregate function is used in the main SELECT clause with a grouping column, the result table must be grouped by all grouping columns, and only the grouping columns for the query to be syntactically correct, as opposed to the optional feature T301 (ISO/IEC, 2016). This grouping convention can be observed in Figure 1 and Table 1 (f, g).

3.2 Practical Examples

In order to demonstrate the usage of the notation in practice, and to demonstrate corresponding SQL clauses with complete examples, we utilize two data demands presented by Taipalus, Siponen, and Vartiainen (2018). We present the query plan formulation in six steps, which correspond to the steps presented in Section 2.2. Additionally, we present the corresponding SQL queries formulated in six steps. It is worth noting that we do not necessarily write the queries in the order presented in Tables 2 and 3, and the tables are presented merely for illustrative purposes. Refer to Appendix 2 for the database schema and business domain.

For Figure 2 and Table 2, consider the data demand "List the names of actors who have acted a role as himself or herself. Sort the results according to surname and then according to first name, both in ascending order."

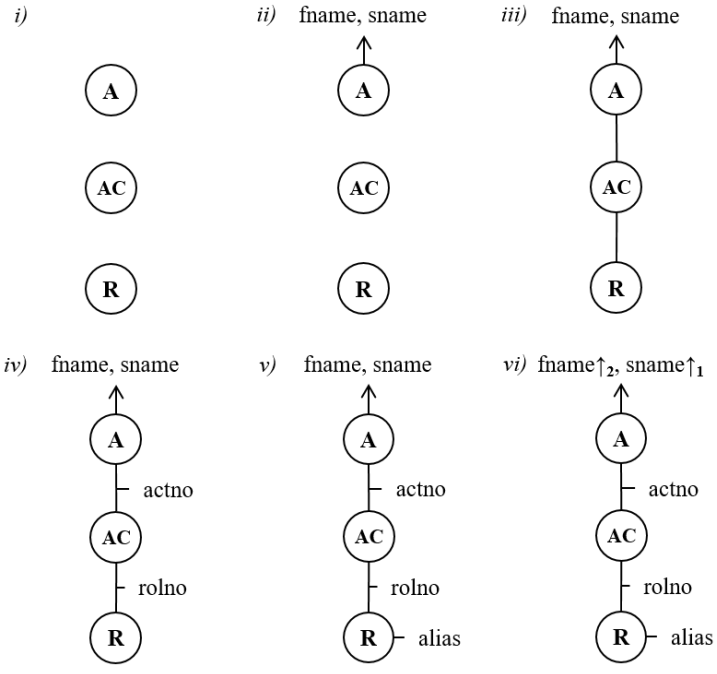


Figure 2. The Iterative Process of a Basic Query Plan Formulation - Table Abbreviations A, AC, and R Stand for Actor, Acts, and Role, Respectively

i)	*ii)*	*iii)*
SELECT FROM actor a, acts ac, role r	SELECT a.fname, a.sname FROM actor a, acts ac, role r	SELECT a.fname, a.sname FROM actor a, acts ac, role r WHERE a. = ac. AND ac. = r.
iv)	*v)*	*vi)*
SELECT a.fname, a.sname FROM actor a, acts ac, role r WHERE a.actno = ac.actno AND ac.rolno = r.rolno	SELECT a.fname, a.sname FROM actor a, acts ac, role r WHERE a.actno = ac.actno AND ac.rolno = r.rolno AND r.alias IN ('Himself', 'Herself')	SELECT a.fname, a.sname FROM actor a, acts ac, role r WHERE a.actno = ac.actno AND ac.rolno = r.rolno AND r.alias IN ('Himself', 'Herself') ORDER BY a.sname ASC, a.fname ASC;

Table 2. The Corresponding SQL Statements for Each Step Presented in Figure 2

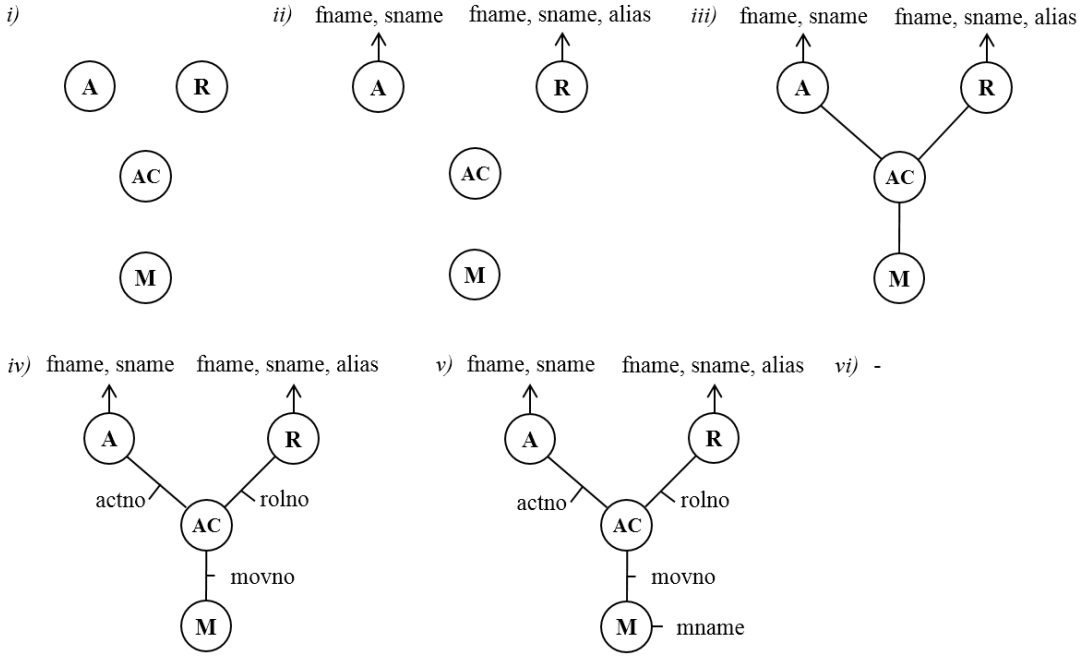


Figure 3. The Iterative Process of a More Complex Query Plan Formulation – Table Abbreviations A, R, AC, and M Stand for Actor, Role, Acts, and Movie, Respectively

i)	*ii)*	*iii)*
SELECT FROM actor a, role r, movie m, acts ac	SELECT a.fname, a.sname, r.fname, r.sname, r.alias FROM actor a, role r, movie m, acts ac	SELECT a.fname, a.sname, r.fname, r.sname, r.alias FROM actor a, role r, movie m, acts ac WHERE a. = ac. AND r. = ac. AND ac. = m.
iv)	*v)*	*vi)*
SELECT a.fname, a.sname, r.fname, r.sname, r.alias FROM actor a, role r, movie m, acts ac WHERE a.actno = ac.actno AND r.rolno = ac.rolno AND ac.movno = m.movno	SELECT a.fname, a.sname, r.fname, r.sname, r.alias FROM actor a, role r, movie m, acts ac WHERE a.actno = ac.actno AND r.rolno = ac.rolno AND ac.movno = m.movno AND m.mname = 'Physics 101';	*(nothing to add)*

Table 3. The Corresponding SQL Statements for Each Step Presented in Figure 3

For Figure 3 and Table 3, consider the data demand "List the names of actors who have acted in the movie *Physics 101*, and list the names of the roles they have played in that movie." For the query plans in Figures 2 and 3, notice how the distances of the nodes from the root node would represent the level of the subqueries.

4. IMPLICATIONS FOR TEACHING

We have identified fours ways of using the notation in teaching. First, when SQL is first taught in the course lectures, query plans can be utilized to explain the logic behind each data demand before writing the query. In our experience, the notation is so simple and intuitive that it can be explained simultaneously to drawing the first query plan. During the drawing process, the teacher can ask students the questions listed in Section 2.2 and draw the plan gradually. The students can be encouraged to plan all queries before writing them for lab assignments or in the final examination.

Second, as the notation separates logic and semantics from syntax, the students can ask the teachers whether their query is planned correctly without focusing on the syntactical aspects of the query. Subsequently, the teachers can point out possible logical errors in the plan, asking questions such as "this plan answers to a different data demand, can you tell me what it is?" This in turn informs the students whether they have understood the data demand and can then focus on the syntax. For example, if we join STORE with EMPLOYEE (see Appendix 2) using *stono*, the result table contains stores with at least one employee working in them. However, the teacher can draw a query plan in which the tables are joined using *empno* and ask the students to explain what the data demand is.

Third, in addition to writing queries in the final examination, query plans may be required. Although this requirement means that the students need to learn an additional notation for the final examination, it might eliminate some errors caused by carelessness, such as missing expressions or ordering from the queries, in addition to forcing the student to reflect on the logic behind the data demand before starting the query writing process.

Fourth, the logic behind joining different tables by different columns in a specific database domain can be practiced in pairs: one student draws the query plan and another student writes the query based on that plan. The exercise can be made more difficult if only the student drawing the query plan is aware of the data demand. We are eager to construct a research setting to see if scientific evidence supports our positive experiences with the notation.

5. CONCLUSION

In this paper, we presented a simple notation for planning complex SQL queries to separate the logic of a data retrieval task from the syntax of SQL and to alleviate the strain a task puts on the query writer's short-term memory. We hope that the paper will encourage other educators to use the notation in their database courses to facilitate the teaching of SQL and to help formulate, understand, and teach more complex queries to mimic the students' future work environments, whether those environments are in the domain of business analytics or software engineering.

6. REFERENCES

Ahadi, A., Prior, J., Behbood, V., & Lister, R. (2016). Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*, 272–277.

Borthick, A. F., Bowen, P. L., Jones, D. R., & Tse, M. H. K. (2001). The Effects of Information Request Ambiguity and Construct Incongruence on Query Development. *Decision Support Systems*, 32, 3-25.

Buitendijk, R. B. (1988). Logical Errors in Database SQL Retrieval Queries. *Computer Science in Economics and Management*, 1(2), 79-96.

Casterella, G. I. & Vijayasarathy, L. (2013). An Experimental Investigation of Complexity in Database Query Formulation Tasks. *Journal of Information Systems Education*, 24(3), 211-222.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377–87.

Codd, E. F. (1972). Relational Completeness of Data Base Sublanguages. *Data Base Systems (Courant Computer Science Symposium 6)*, Prentice-Hall.

de Jong, T. (2010). Cognitive Load Theory, Educational Research, and Instructional Design: Some Food for Thought. *Instructional Science*, 38, 105–134.

Hu, M., Winikoff, M., & Cranefield, S. (2012). Teaching Novice Programming using Goals and Plans in a Visual Notation. *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123 (ACE '12)*, Darlinghurst, Australia, 43-52.

ISO/IEC. (2016). ISO/IEC 9075-2:2016, *SQL - Part 2: Foundation*.

Randolph, G. B. (2003). The Forest and the Trees: Using Oracle and SQL Server Together to Teach ANSI-Standard SQL. *Proceedings of the 4th Conference on Information Technology Curriculum (CITC4)*, 234–236.

Smelcer, J. B. (1995). User Errors in Database Query Composition. *International Journal of Human-Computer Studies*, 42(4), 353–381.

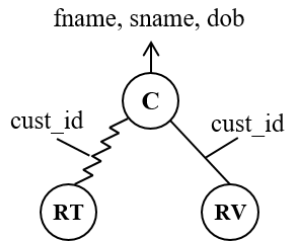
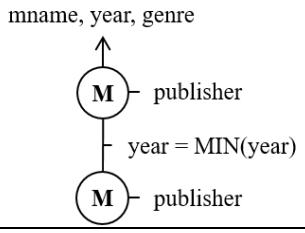
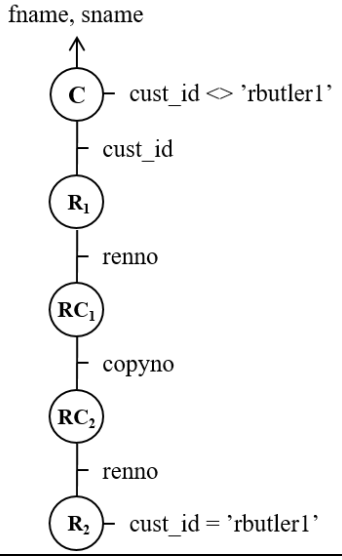
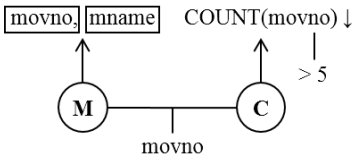
Taipalus, T., Siponen, M., & Vartiainen, T. (2018). Errors and Complications in SQL Query Formulation. *ACM Transactions on Computing Education,* 18(3), Article 15.

AUTHOR BIOGRAPHY

Toni Taipalus is a teacher at the University of Jyvaskyla. He

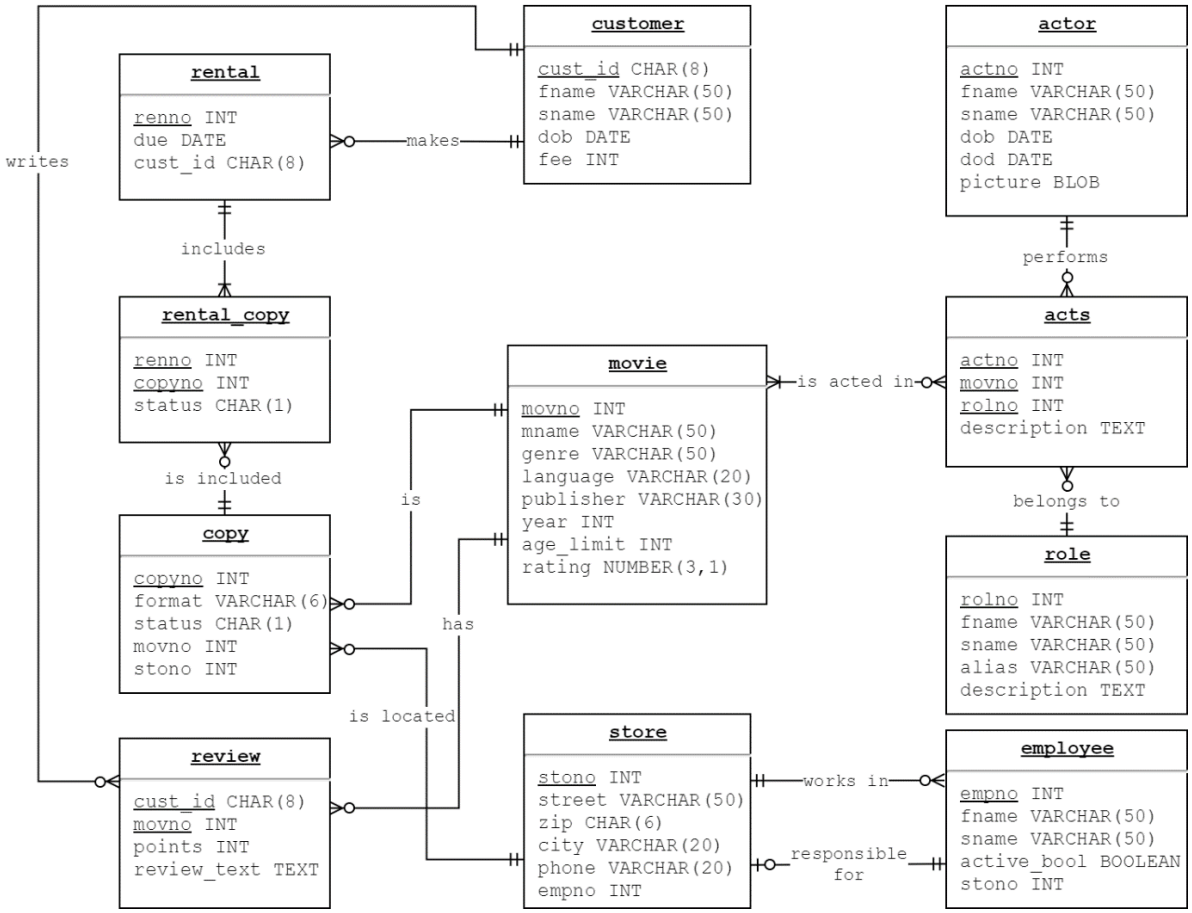
teaches databases, data management, application programming, and system development. His research interests are in the pedagogical aspects of query languages, data models, and agile software development.

APPENDIX 1: EXAMPLE QUERY PLANS

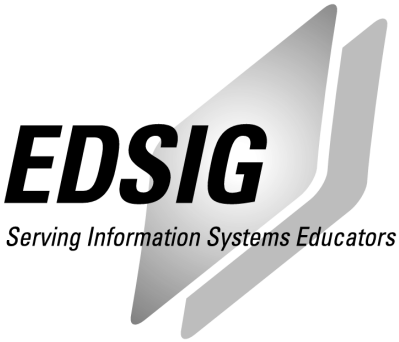
Query (Taipalus, Siponen, and Vartiainen, 2018)	Query plan
SELECT c.fname, c.sname, c.dob FROM customer c WHERE NOT EXISTS (SELECT * FROM rental rt WHERE c.cust_id = rt.cust_id) AND EXISTS (SELECT * FROM review rv WHERE c.cust_id = rv.cust_id);	
SELECT mname, year, genre FROM movie WHERE publisher =‘Goldeneye BC’ AND year = (SELECT MIN(year) FROM movie WHERE publisher =‘Goldeneye BC’);	
SELECT c.fname, c.sname FROM customer c, rental r1, rental_copy rc1, rental_copy rc2, rental r2 WHERE c.cust_id = r1.cust_id AND r1.renno = rc1.renno AND rc1.copyno = rc2.copyno AND rc2.renno = r2.renno AND r2.cust_id = ‘rbutler1’ AND c.cust_id <> ‘rbutler1’;	
SELECT m.movno, m.mname, COUNT(c.movno) AS total FROM movie m, copy c WHERE m.movno = c.movno GROUP BY m.movno, m.mname HAVING COUNT (c.movno) > 5 ORDER BY total DESC;	

APPENDIX 2: THE DATABASE SCHEMA

This appendix contains a database schema (Taipalus, Siponen, and Vartiainen, 2018) to be used in conjunction with the examples in Sections 3.2 and 4 and Appendix 1.



Information Systems & Computing
Academic Professionals



STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 2574-3872