# DBSnap-Eval: Identifying Database Query Construction Patterns

Yasin N. Silva
Loyola University Chicago
ysilva1@luc.edu

Alexis Loza
Arizona State University
aloza9@asu.edu

Humberto Razente
Universidade Federal de Uberlândia
humberto.razente@ufu.br

## ABSTRACT

Learning to construct database queries can be a challenging task because students need to learn the specific query language syntax as well as properly understand the effect of each query operator and how multiple operators interact in a query. While some previous studies have looked into the types of database query errors students make and how the availability of expected query results can help to increase the success rate, there is very little that is known regarding the patterns that emerge while students are constructing a query. To be able to look into the process of constructing a query, in this paper we introduce DBSnap-Eval, a tool that supports tree-based queries (similar to SQL query plans) and a block-based querying interface to help separate the syntax and semantics of a query. DBSnap-Eval closely monitors the actions students take to construct a query such as adding a dataset or connecting a dataset with an operator. This paper presents an initial set of results about database query construction patterns using DBSnap-Eval. Particularly, it reports identified patterns in the process students follow to answer common database queries.

## CCS CONCEPTS

• **Social and professional topics** → **Computing education**; Information systems education.

## KEYWORDS

Relational algebra, SQL, block-based system, database systems, database queries

**ACM Reference format:**

Yasin Silva, Alexis Loza, Humberto Razente. 2022. DBSnap-Eval: Identifying Database Query Construction Patterns. In *Proceedings of ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'22). ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3502718.3524822

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*ITiCSE 2022, July 8–13, 2022, Dublin, Ireland*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524822

## 1 Introduction

Database queries are the main mechanisms to interact with database systems. Multiple querying languages have been proposed for relational databases such as relational calculus (RC), relational algebra (RA), and the Structured Query Language (SQL). A good understanding of fundamental languages like RA provides an excellent basis to learn SQL, which is also supported in multiple Big Data systems due to its expressive power [20]. Learning how to construct a database query can be challenging. Not only does one have to learn the specific syntax of a query language, but also the data processing logic associated with each operator and the ways multiple operators interact in a query. The study of how learners create queries and the methodologies that can help students to better understand database querying has received relatively little attention in previous work. While some previous studies have reported the types of query errors students make, potential reasons behind these errors, and how the availability of expected query results can help to increase student success rate, there is very little that is known regarding the process students follow to construct a query and the way this information can be used to help students and educators.

A core goal of this paper is to shed light on the patterns that emerge during query creation. To facilitate this, we implemented DBSnap-Eval, an extended version of DBSnap [10, 11]. DBSnap is a block-based querying tool that enables learners to specify database queries by dragging and connecting visual blocks (datasets and operators). DBSnap supports the construction of query trees that represent the core structure of relational algebra and SQL queries. A benefit of using this tool is that it helps to separate the syntax and semantics of a query. We extended DBSnap adding a monitoring module that obtains some initial non-identifying information about the student and monitors the steps or actions the student completes to create a query. We used DBSnap-Eval to collect information about ten common types of queries. The main contributions of the paper are:

- We introduce DBSnap-Eval [21], an open-source tool to monitor the query construction process. This tool supports tree-based query representations and can be used by other researchers to perform further analysis of query construction patterns.
- We present an initial set of results about database query construction patterns (with 712 submitted query answers from 78 database class students and manual evaluation of query correctness). This includes: (1)

query correctness rates across different query types, (2) high-level temporal properties, e.g., average times to complete specific query types, and (3) query construction details such as the percentage of time used on specific operators, and the number and type of steps used to build each type of query.

- We highlight some identified patterns that could help educators to better teach database querying (in DBSnap and potentially other tools), e.g., common student misunderstandings during query construction and approaches that could increase the likelihood to produce correct answers.

## 2 Related Work

***Learning database query languages***. Some of the few studies in this area focused on understanding the types of errors students make. Taipalus et al. investigated the types of errors that students make in SQL exercises (particularly errors that are likely to be left uncorrected) and identified concepts that are harder for the students to learn, such as joins and aggregations [12]. This study found that syntax errors are less likely to persist than logical errors. Ahadi et al. identified the most common semantic mistakes students make while writing SQL queries and suggested reasons behind these mistakes [2]. This study reported that the most common errors were related to omitting needed query constructs. Migler and Dekhtyar reported the average number of attempts students made while writing SQL queries and the associated success rates for different query types [7]. In most of the evaluated exercises in this work, students were able to access the expected query output. Some studies applied data mining techniques to better identify and predict database query errors. Lino et al. used clustering and decision trees to classify different types of SQL errors and identify the most frequently committed ones [6]. Ardiansah et al. built a model to identify possible logic SQL errors and report them as warnings [3]. This model was integrated into a web-based SQL compiler that can be used by students. The work by Ahadi et al. evaluated several SQL syntactical errors and developed a rule-based classifier to predict student success in writing queries [1]. Kleerekoper et al. developed and investigated the effectiveness of an online testing tool (SQL Tester) for SQL [4]. This work found a strong correlation between the number of practice tests and the test scores. Fully evaluating whether a query is correct or not is challenging. Many of the previous studies, including [2, 3, 4, 6, 7], consider a query to be correct if it generates the expected output. It is well-known, however, that incorrect queries can generate the correct answer on a specific state of the data. Our work differs from previous contributions in that we focus on better understanding the process in which queries are constructed. Our work also includes manual evaluation of query correctness.

***Block-based querying systems***. Multiple tools have been proposed to simplify the process of specifying database queries. The query- by-example model [15] is an early approach that supported visual elements to write queries. Several contributions enabled the specification of queries using a set of icons. The SQL Visualiser [8] allows the specification of simple queries by dropping icons representing data items into a box. The system then transforms the visual specification into an SQL query. iDFQL [5] and RALT [17] use icons to represent relational algebra operators and make use of flow diagrams to specify the sequence of operations in a query. More recently, several contributions enabled constructing queries using blocks, e.g., SQLsnap [16], BlocklySQL [9], Bags [14], DBSnap [10, 11] and DBSnap++ [19]. Several of these tools are extensions of Snap! [13] and Blockly [18], block-based tools for specifying standard computer programs. In our study, we decided to extend a block-based querying tool (DBSnap) as the basis of our query construction evaluation tool because it enables learners to focus on the query logic instead of its syntax.

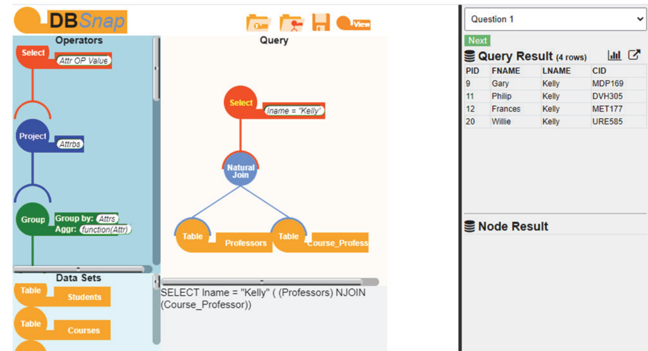


**Figure 1: DBSnap-Eval interface and query tree example**

## 3 DBSnap-Eval and Data Collection

In order to gain insights about the process that students follow while building queries, we implemented DBSnap-Eval extending DBSnap [10, 11]. DBSnap-Eval contains all the interface elements of DBSnap (see Fig. 1): dataset and block palettes (left), query area (middle), and results panels (right). Students build queries, like the one shown in Fig. 1, by dragging and connecting blocks. The results are dynamically generated after each query construction action (e.g., adding a new operator). DBSnap uses query trees to represent queries. Query trees have been used by many educators and textbooks as an intuitive way to represent relational algebra and SQL queries. They show the organization of the different operators and the way intermediate results flow (bottom-up) in the query pipeline. DBSnap-Eval extends DBSnap by adding support for surveys, enabling the selection of questions and submission of answers, and a monitoring component that records a log of the query actions completed by the student. The actions include moving a block from a palette to the query area (*Move*), updating the fields of a block (*Update*), and connecting (*Connect*), disconnecting (*Disconnect*) and deleting (*Delete*) blocks. For every action, the tool records the time and type of the action, and the involved blocks. At query submission time, it also records the query representation and output. The actions executed by a student in DBSnap correspond to writing specific query elements in the text version of a query (e.g., moving/updating a Select block corresponds to writing the Select keyword and its parameters).

**Table 1 Database schema and list of database query questions**

| Database Schema | | |
|---|---|---|
| **Students** (SID, LName, FName, Level, Age), **Courses** (CID, CName), **Professors** (PID, LName, FName), **Course_Student** (CID, SID), **Course_Professor** (CID, PID), **Student_Club** (studentID, clubName), **StudentsA** and **StudentsB** have the same schema as Students | | |
| **Query** | **Description + {Needed Number of Actions} - SQL** | **Operators** |
| Sel | Find the students that are older than 23 {3}<br>SELECT * FROM Students WHERE Age>23 | Selection |
| Proj | Find the first and last name of every professor {3}<br>SELECT FName, LName FROM Professors | Projection |
| Int | Find the students that are both Student A's and Student B's {3}<br>SELECT * FROM StudentsA INTERSECT SELECT * FROM StudentsB | Intersection |
| Join | Find the list of students and the courses they take {3}<br>SELECT * FROM Students NATURAL JOIN Course_Student | Natural Join |
| SJoin | Find the courses that professors with the last name 'Kelly' teach {5}<br>SELECT * FROM Professors NATURAL JOIN Course_Professor WHERE Professors.LName = 'Kelly' | Selection, Natural Join |
| SPJ | Find the student ID, the last name, and the course ID of 'Junior' students who are between 18 and 20 years old inclusive {7}<br>SELECT SID, LName, CID FROM Students NATURAL JOIN Course_Student WHERE Level = 'Junior' and Age>=18 and Age<=20 | Selection, Projection, Natural Join |
| PJoin | Find the list of courses (CID, CName) and the professors teaching them (PID, LName) {5}<br>SELECT CID, CName, P.PID, P.LName FROM Professors AS P NATURAL JOIN Course_Professor | Projection, Natural Join |
| Theta | Find the students participating in clubs {4}<br>SELECT * FROM Students, Student_Club WHERE SID = studentID | Theta Join |
| Aggr | Find the avg. age of students for every year level {4}<br>SELECT Level, AVG(Age) FROM Students GROUP BY Level | Group By |
| JAgg | How many students are in every course (CName) {6}<br>SELECT CName, COUNT(CS.SID) FROM Courses as C NATURAL JOIN Course_Student as CS GROUP BY C.CID, CName | Natural Join, Group By |

In our study, we used DBSnap-Eval to collect a total of 712 submissions from 78 students taking a database class. We used 10 of the most common types of database queries (schema and queries shown in Table 1). Each student also completed a survey providing information about gender, previous database experience, and previously completed database classes. Students were trained on the visual elements of DBSnap-Eval before they participated in this study. They were also informed about the nature of this study, how the querying steps they would complete were going to be monitored (without recording identifying information like student name or IP address), and how the collected data was going to be used (aggregated report). Participation was optional but most students participated. Unlike most of the previous work in this area (which consider that a query is correct when it generates the expected output), we manually evaluated the correctness of each submitted query. Manually evaluating queries was a time-consuming task and limited the number of students and submissions we included in our study. On the other hand, this step provided an accurate assessment of query correctness as incorrect queries can easily generate the expected output on a given state of the dataset.

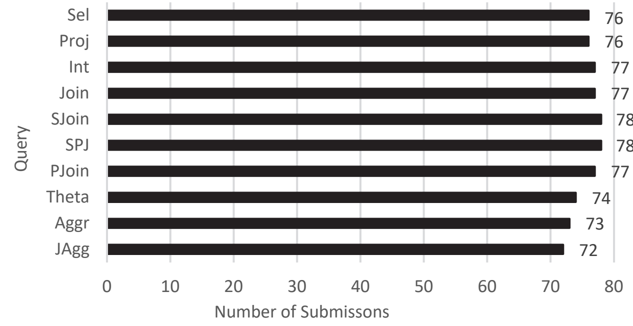


**Figure 2: The total number of submissions per query**

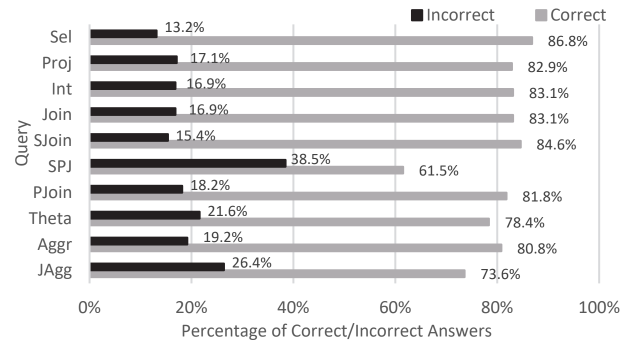


**Figure 3: Correctness rate per query**

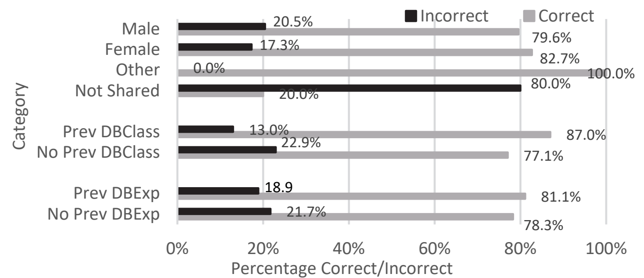


**Figure 4: Correctness rate for questionnaire categories**

# 4 Results

In this section, we present the results of various data analysis tasks. Specifically, we explore aspects related to query success rates, temporal properties, and query construction patterns. Out of the 78 students who submitted solutions, 77% (60) were male, 21% (16) female, 1% (1) selected other gender, and 1%(1) did not share this information. 26% (20) of the students had previously taken a database class and 49% (38) had some prior basic exposure or experience with databases, e.g., internship, work, etc. (not necessarily using querying languages). Fig. 2 shows the number of submissions received for each question. We received at least 72 submissions for each query question.

***Query correctness rate per query type and questionnaire category***. Fig. 3 shows the correctness rate for each query question. Across all the queries, between 62% to 87% of the submissions were correct. In general (and as expected), the

correctness rate is lower in queries with larger number of operators. The average rate for queries with one, two, and three operators was 82.5%, 80%, and 61%, respectively. This, however, is not a strict or uniform rule. We can observe, for instance, that queries with aggregation and Theta join operations had smaller success rates. We also observe that while the difference between queries with one and two operators is small, the difference between queries with two and three operators is significantly larger. Since our study focused on relatively simpler queries in an introductory database class, we did not evaluate queries with large number of operators. Fig. 4 shows the correctness rates for the categories used in the survey across all queries. We found that female students (82.7%) performed slightly better than male students (79.6%) and that the success rate for students who took a database course previously (87%) was relatively higher than the one for students who were taking their first database class (77%).

***Query correctness vs. time and number of actions***. Fig. 5 and Fig. 6 show the time and number of actions that students took to complete each type of query, respectively. Queries with a single operator are represented with black dots while the ones with two or more with gray dots. As reported in previous work, we found that, in general, queries with aggregations and joins have smaller success rates. We found, however, significant differences between the query with aggregation only (*Aggr*) and the one that combines aggregation and join (*JAgg*). The *Aggr* query took a time (1m 48s) similar to other single-operator queries (1m 33s) but significantly smaller than *JAgg* (2m 20s). Likewise, *Aggr* took a number of actions (9.4) that is similar to other single-operator queries (average: 12.8), but significantly smaller than that of *JAgg* (18.6). Observe also that while *Aggr* and *JAgg*, required a minimum of 4 and 6 actions, respectively, students took on average 9 and 19 actions to complete them. These results seem to indicate that the concept of aggregation by itself may not be much more complex than other operators such as selection and projection but that students find the combination of join and aggregation particularly more challenging. We also observe that in general, students took significantly more actions to complete a query than needed. While the smallest number of actions to correctly build a query (see Table 1) ranged from 3 (e.g., *Sel*, *Proj*, and *Int*) to 7 (*SPJ*), the actual number of completed actions ranged from around 8 (*Sel*, *Proj*, *Int*, *Aggr*), to around 24 (*PJoin*, *Theta*, *SJoin*). This highlights the value of interactive querying interfaces that show students the results of a query as it is being constructed. We also observe that all the queries with join operators take more time and actions than other queries. The divide is even more evident in terms of number of actions. While *SJoin* takes more time and actions than other join queries, this may be related to the fact that in most cases, this was the first multi-operator query that students answered (queries could be answered in any order).

***Time spent in top-3 query blocks***. Figuring out how much time students spend on a query block can highlight the parts of the query they struggled the most with. Fig. 7 shows the top-3 query blocks (datasets and operators) for each query type based on the time spent with that type of block while constructing the query.

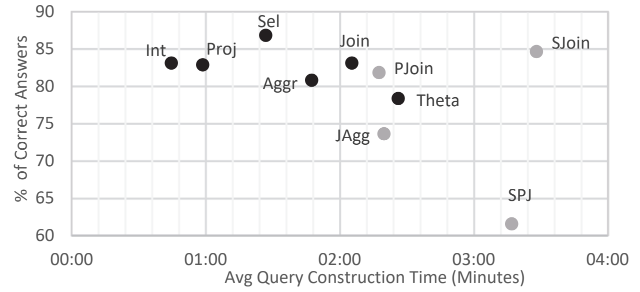


**Figure 5: Query construction time vs. correctness**

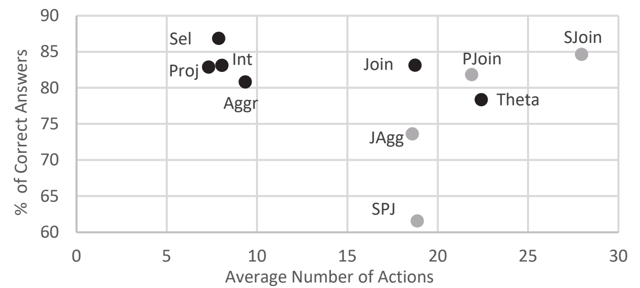


**Figure 6: Number of actions vs. correctness**

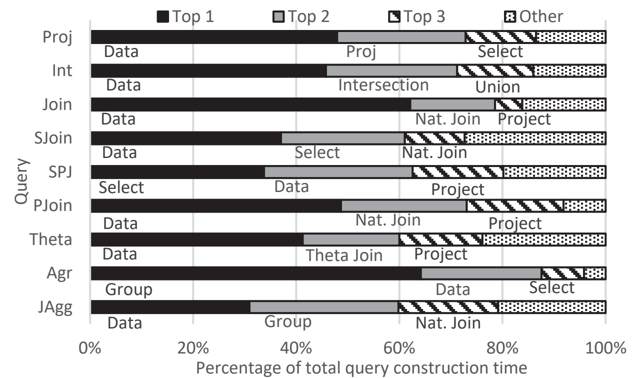


**Figure 7: Percentage of time spent in top 3 query blocks**

Two general and somewhat expected observations are that, in most queries, datasets are the most common blocks and the required operators to solve the query are also among the top-3 operators. There are, however, interesting insights we can infer from the presence of operators that were not needed to answer the queries. There are four specific cases. (1) Selection-Projection: In *Proj*, the query did not require the selection operator, however this block was a top-3 block. This seems to be a common confusion since both operators can be seen as types of "filtering" operators. Selection filters the rows of a table while projection filters the columns. (2) Set operators: The *Int* query required the intersection operator, but the union operator was also commonly used here. This shows that not properly recognizing the differences among set operations is a common mistake. (3) Grouping-Selection: In *Aggr*, the query only needed the grouping operator but the select operator was a top-3 block.

This seems to indicate that some students want to express the grouping predicate as a selection predicate. This confusion may be related to the fact that both operators usually generate an output that is significantly smaller than the input data. Some students fail to recognize that these operators reduce the data cardinality in very different ways. (4) Join-Projection: This was an unexpected finding. Both, *Join* and *Theta*, required a single join operator. However, in both cases, the projection operator was among the top-3 blocks. The reason may be related to the fact that students learn that the projection operator controls the set of attributes that will be included in the output. Students understand that the output of a join query should include attributes from different tables. They seem to fail to understand, however, that projection by itself cannot include attributes of datasets that are not included in the query. These four identified cases represent potential common sources of confusion.

***Number of different types of actions per query type.*** Fig. 8 and Fig. 9 show the number of actions for each query among correct and incorrect responses, respectively. Across all submissions and query types, we can observe that the *Move* and *Connect* actions are the most common ones. *Disconnect* and *Delete* actions are less common but are present in both correct and incorrect submissions. *Disconnect* and *Delete* represent corrections made during the query construction process. *Update* actions are less common because only certain operators (the ones with predicate fields) allow this action. We observe that the number of actions between correct and incorrect submissions are, in general, similar. There are, however, two outliers: *Join* and *JAgg*. In *Join*, the number of *Move* and *Connect* actions in incorrect submissions are 49% and 82% larger than those in correct ones, respectively. We believe this is the case because this was the first multi-table query most students answered. The practice answering this question probably helped in subsequent join queries. In *JAgg*, the number of *Connect* and *Disconnect* actions in incorrect submissions are 2.4x and 4x larger than those in correct ones, respectively. This result is aligned with our previous finding about the increased difficulty of combining joins and aggregations.

***Closer look into the first five minutes of query construction.*** Fig. 10 and Fig. 11 show the average number of different types of actions during the first 5 minutes (in 1-min intervals) for correct and incorrect *Join* query submissions, respectively. Similar trends were observed in other queries. We notice that the number of actions changes over time with most actions happening in the first 2 minutes. While both correct and incorrect submissions show significant levels of activity, we observe some differences. In the correct submissions, the *Move* and *Connect* actions dominate in the entire time range. The number of corrective actions (*Disconnect* and *Delete*) are overall small but tend to increase over time. In the incorrect submissions, the *Move* and *Connect* actions are the most common ones but the number of corrective actions is significantly larger than in correct submissions. These results indicate that, while a trial-and-error process appears to be used in both correct and incorrect queries, this process is more active in incorrect queries.

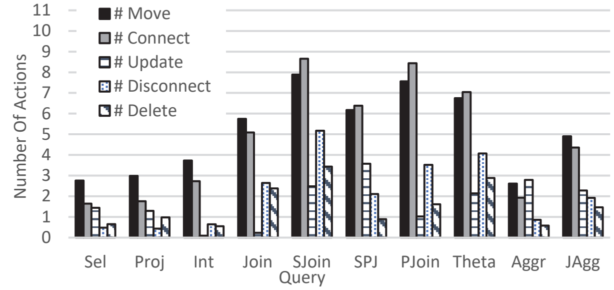


**Figure 8: No. of actions per question (correct answers)**

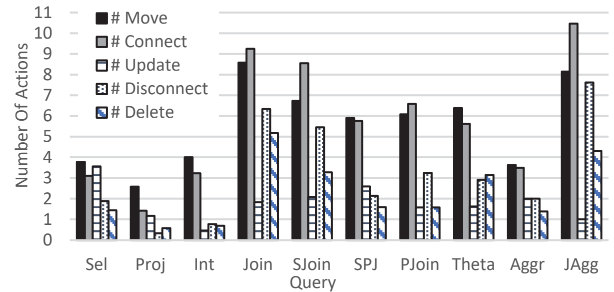


**Figure 9: No. of actions per question (incorrect answers)**

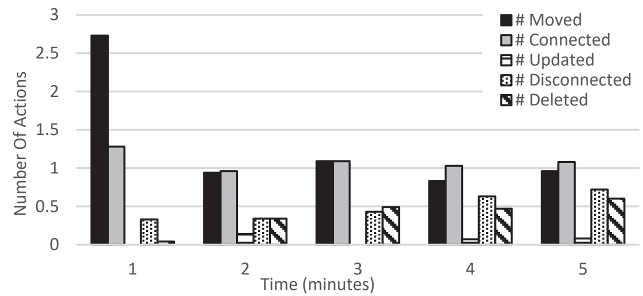


**Figure 10: First 5 minutes of *Join* (correct answers)**

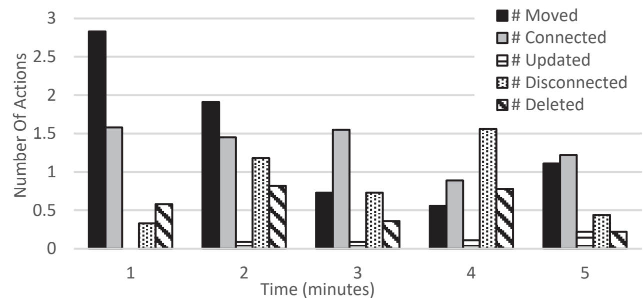


**Figure 11: First 5 minutes of *Join* (incorrect answers)**

***Data-first vs operator-first.*** Knowing how to start a query can be quite challenging when learning database query languages. We analyzed the collected data to identify early actions that can help students build correct queries. Fig. 12 shows the percentage of correct and incorrect queries comparing submissions that started the query construction process with dataset blocks vs.

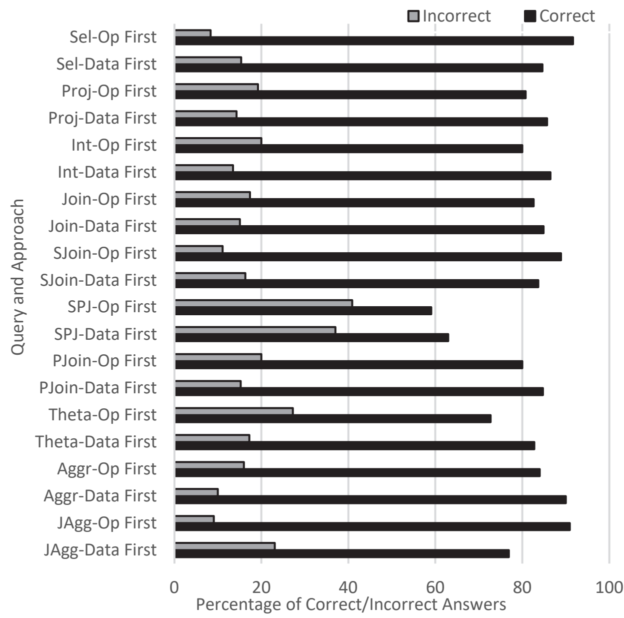


**Figure 12: Operator-first vs. Data-first**

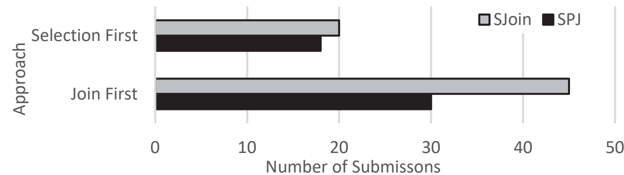


**Figure 13: Selection-first vs. Join-first**

submissions that started with operator blocks. This figure shows that in most query types (seven out of ten: *Pro*, *Int*, *Join*, *SPJ*, *PJoin*, *Theta*, and *Aggr*), submissions that started with the database blocks have higher success rate than the ones that started with the operator blocks. While the advantage is not large (e.g., the success rate increases by 10% in *Theta* and 6.5% in *Int*), these results seem to indicate that, for certain queries, starting the process thinking about what datasets (tables) are necessary may be a beneficial query construction strategy.

***Query optimization opportunities.*** Our evaluation was performed in an introductory database class where there were no expectations on generating optimized query plans. DBSnap being based on query-tree representations, however, is a good platform to introduce the concepts of query optimization. One of the key ideas in query optimization is the use of query plans that filter large amounts of data early in the query execution process. This way later operators (higher in the query tree) will work on reduced amounts of data. In our study, we observed that most students wrote queries that in practice would correspond to potentially inefficient plans. Fig. 13 shows that for queries *SJoin* and *SPJ*, 66% of the submitted queries executed the join first while only 34% of the queries executed the selection operator first (in general, the latter is expected to be more efficient). This highlights the opportunity to introduce concepts of query optimization using simple queries in introductory database classes.

## 5 Discussion, Limitations and Extensions

Our study of patterns that emerge while students write queries revealed multiple results that could be used by database educators to better teach query languages. The reported results include: (1) queries that combine aggregation and join operators are particularly challenging for students, (2) there are common pairs of operators where one operator is incorrectly used instead of the other one (Selection-Projection, Set operators, Grouping-Selection, Join-Projection), (3) the trial and error process is actively used by query language learners, using interfaces that facilitate this process by dynamically showing the results as the query is constructed can be specially beneficial to students, (4) starting a query thinking about the data needed to answer the query may be a beneficial strategy in certain cases, (5) there are opportunities to introduce query optimization concepts even with simple queries. Given the limited number of studies in this area, this field can benefit from more empirical evidence.

This experience report has some limitations. It only includes students from a single instructor. This may have influenced some of the identified query construction patterns. Also, while the manual evaluation of query correctness eliminated the possibility of false-positives (incorrect queries with correct output), this also limited the number of submissions included in our study. We plan to evaluate more submissions from multiple instructors and report additional findings in the future.

While DBSnap-Eval provides researchers with some features not supported in other tools, e.g., the ability to monitor query construction *while* queries are constructed, it can be extended in multiple ways. For instance, it can be adapted to support other languages such as domain calculus and a text-oriented version of SQL. It can also be adapted to detect significant deviations from the correct answers and suggest corrective actions. It could also be used to study additional query construction properties, e.g., (1) identify commonly occurring correct/incorrect submissions for a given query, (2) study complex operators (e.g., division) and multi-operator queries, (3) study the use of views to construct complex queries, (4) identify rates in which incorrect query submissions produce the correct output, and (5) identify cross-query trends among the queries submitted by a student.

## 6 Conclusion

Very little is known about *how* query language learners write database queries. Identifying query construction patterns and common errors in the process of building queries can help educators to better teach query languages. To this end, in this paper we introduce DBSnap-Eval, an open-source system to monitor the process of creating tree-based database queries (similar to the query evaluation trees used by many database textbooks and educators) and present an initial set of identified query construction patterns. A goal of making DBSnap-Eval publicly available is to enable the research community to perform further studies regarding the query construction process.

# REFERENCES


[1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and Its Application to Predicting Students' Success. In ACM Technical Symposium on Computing Science Education (SIGCSE'16) (Memphis, TN). ACM, 401–406. https://doi.org/10.1145/2839509.2844640.

[2] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. 2016. Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. In ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'16) (Arequipa, Peru). ACM, 272–277. https://doi.org/10.1145/2899415.2899464.

[3] Jevri Tri Ardiansah, Aji Prasetya Wibawa, Triyanna Widyaningtyas, and Okazaki Yasuhisa. 2017. SQL Logic Error Detection by Using Start End Mid Algorithm. Knowledge Engineering and Data Science 1, 1 (2017), 33–38. https://doi.org/10.17977/um018v1i12018p33-38.

[4] Anthony Kleerekoper and Andrew Schofield. 2018. SQL Tester: An Online SQL Assessment Tool and Its Impact. In ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'2018) (Larnaca, Cyprus). ACM, 87–92. https://doi.org/10.1145/3197091.3197124.

[5] Ana Paula Appel, Elaine Quintino da Silva, Caetano Traina Jr., and Agma J. M. Traina. 2004. iDFQL: a query-based tool to help the teaching process of the relational algebra. In World Congress on Engineering and Technology Education (WCETE). 179–184.

[6] Adriano Lino, Álvaro Rocha, Luís Macedo, and Amanda Sizo. 2019. Application of clustering-based decision tree approach in SQL query error database. Future Generation Computer Systems 93 (2019), 392–406. https://doi.org/10.1016/j.future.2018.10.038.

[7] Andrew Migler and Alex Dekhtyar. 2020. Mapping the SQL Learning Process in Introductory Database Courses. In ACM Technical Symposium on Computer Science Education (SIGCSE'20) (Portland, OR). ACM, 619–625. https://doi.org/10.1145/3328778.3366869.

[8] George Obaido, Abejide Ade-Ibijola, and Hima Vadapalli. 2019. Generating SQL Queries from Visual Specifications. In ICT Education. Springer, 315–330. https://doi.org/10.1007/978-3-030-05813-5_21.

[9] Nicolai Pöhner, Timo Schmidt, André Greubel, Martin Hennecke, and Matthias Ehmann. 2019. BlocklySQL: A New Block-Based Editor for SQL. In Workshop in Primary and Secondary Computing Education (WiPSCE'19) (Glasgow, Scotland). ACM. https://doi.org/10.1145/3361721.3362104.

[10] Yasin N. Silva and Jaime Chon. 2015. DBSnap: Learning Database Queries by Snapping Blocks. In ACM Technical Symposium on Computer Science Education (SIGCSE'15) (Kansas City, MO). ACM, 179–184. https://doi.org/10.1145/2676723.2677220.

[11] Yasin N. Silva and Jaime Chon. 2015. Querying databases by snapping blocks. In International Conference on Data Engineering (ICDE) (Seoul, South Korea). IEEE, 1472–1475. https://doi.org/10.1109/ICDE.2015.7113404.

[12] Toni Taipalus and Piia Perälä. 2019. What to Expect and What to Focus on in SQL Query Teaching. In ACM Technical Symposium on Computer Science Education (SIGCSE'19) (Minneapolis, MN). ACM, 198–203. https://doi.org/10.1145/3287324.3287359.

[13] UC Berkeley. 2013. Snap! Website. http://snap.berkeley.edu/.

[14] Jason Gorman, Sebastian Gsell, and Chris Mayfield. 2014. Learning Relational Algebra by Snapping Blocks. In ACM Technical Symposium on Computer Science Education (SIGCSE'14) (Atlanta, GA). ACM, 73–78. https://doi.org/10.1145/2538862.2538961.

[15] Moshé M. Zloof. 1975. Query by Example. In Proceedings of the National Computer Conference and Exposition (AFIPS'75) (Anaheim, CA). ACM, 431–438. https://doi.org/10.1145/1499949.1500034.

[16] Eckart Modrow. 2018. SQLsnap!. Retrieved May 18, 2020 from http://snapextensions.uni-goettingen.de/.

[17] Pritam Mitra. 2009. Relational Algebra Learning Tool. Technical Report, Dept. Computing, Imperial College London. Retrieved May 18, 2020 from https://www.doc.ic.ac.uk/~pjm/teaching/student_projects/pm105_report.pdf.

[18] Blockly. 2012. Blockly: A JavaScript library for building visual programming editors. Retrieved from https://developers.google.com/blockly.

[19] Yasin N. Silva, Anthony Nieuwenhuyse, Thomas G. Schenk, and Alaura Symons. 2018. DBSnap++: Creating Data-Driven Programs by Snapping Blocks. In ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'2018) (Larnaca, Cyprus). ACM, 170–175. https://doi.org/10.1145/3197091.3197114.

[20] Yasin N. Silva, Isadora Almeida, and Michell Queiroz. 2016. SQL: From Traditional Databases to Big Data. In ACM Technical Symposium on Computing Science Education (SIGCSE'16) (Memphis, Tennessee). ACM, 413–418. https://doi.org/10.1145/2839509.2844560.

[21] Yasin N. Silva. 2022. DBSnap-Eval source-code. Retrieved from https://ysilva.cs.luc.edu/dbsnap/files/dbsnap-eval.zip.