DBSnap: Learning Database Queries by Snapping Blocks

Yasin N. Silva
Arizona State University
ysilva@asu.edu

Jaime Chon
Arizona State University
jchon@asu.edu

ABSTRACT

A significant amount of recent research in Computer Science Education has focused on studying block-based programming. In this approach, computer programs are created by connecting blocks and the blocks' shapes determine the permitted connections. The focus is on the program's logic instead of its syntax. This paper introduces DB*Snap*, a web-based application to build database queries, particularly relational algebra queries, by snapping blocks. DB*Snap* fully supports the construction of intuitive database query trees, which is one of the most effective approaches to teach database queries. DB*Snap* is also highly dynamic and shows the query results as the query is being built. The user can also inspect, at any time, the intermediate results of any query node. This paper presents DB*Snap*'s design and implementation details, an evaluation of its effectiveness as a learning environment, and a thorough comparison with alternative ways to teach query languages. DB*Snap* is publicly available and aims to have the same transformational effect on database learning as previous block-based systems had on traditional programming learning.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Design, Experimentation

Keywords

Databases curricula; query languages; relational algebra

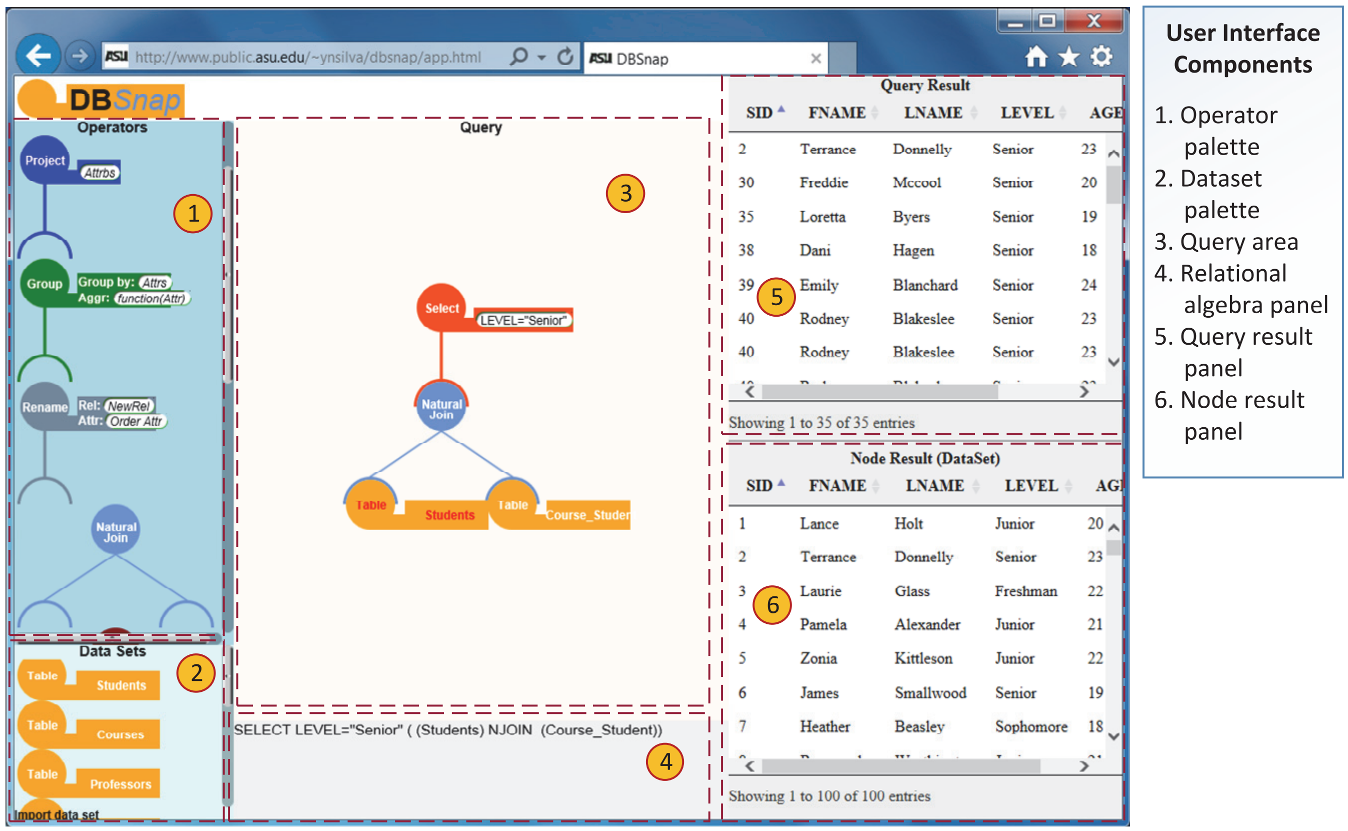
1. INTRODUCTION

Learning computer languages is a challenging endeavor that requires students not only to focus on a logical specification of instructions to be executed by a computer but also on a fully correct syntactical representation of the instructions. The latter component frequently frustrates students because a program will not execute, even if it has a correct logic, due to minor syntactical errors. Recently, a significant amount of research in the Computer Science Education community has focused on the design and development of block-based programming environments, e.g., Mindstorm [6], Scratch [7], Blockly [8], Snap! [11], and App Inventor [15]. The key idea is to enable students to create programs by just dragging and connecting blocks and thus, the focus is on the program's logic instead of its syntax. This idea has revolutionized the way instructors can teach computer programming and has also enabled the introduction of programming concepts to younger students.

This paper introduces DB*Snap*, a web-based application to build database queries, particularly relational algebra queries, by snapping blocks. A key feature of DB*Snap* is the representation of queries as trees. This representation has been one of the most effective approaches to teach database queries because it intuitively represents the organization of the query operators and the flow of intermediate results. This feature and the fact that DB*Snap* dynamically shows the intermediate results as the query is being built are two of several differences between DB*Snap* and the work by Gorman et al. that recently introduced a block-based system to build relational algebra expressions [5]. This paper presents the design, implementation details, and evaluation of DB*Snap*. The main contributions of the paper are:

- The introduction of DB*Snap*, a web-based and dynamic application that enables the construction of highly intuitive database query trees.
- A detailed description of DB*Snap*'s design and implementation guidelines. Our goal is to enable other researchers to extend or customize DB*Snap*.
- The evaluation of DB*Snap*'s effectiveness as a learning environment and a detailed comparison with alternative systems and tools to teach database query languages.
- The public availability of DB*Snap* to be used by any instructor or student and a set of query examples. DB*Snap* aims to have a transformational effect on database learning.

The rest of the paper is organized as follows. Section 2 presents the design details of DB*Snap*. Section 3 describes the details of implementing DB*Snap* as a web application.



```
SELECT LEVEL="Senior" ( (Students) NJOIN (Course_Student))
```

Query Result

SID	FNAME	LNAME	LEVEL	AGE
2	Terrance	Donnelly	Senior	23
30	Freddie	Mccool	Senior	20
35	Loretta	Byers	Senior	19
38	Dani	Hagen	Senior	18
39	Emily	Blanchard	Senior	24
40	Rodney	Blakeslee	Senior	23
40	Rodney	Blakeslee	Senior	23

Showing 1 to 35 of 35 entries

Node Result (DataSet)

SID	FNAME	LNAME	LEVEL	AG
1	Lance	Holt	Junior	20
2	Terrance	Donnelly	Senior	23
3	Laurie	Glass	Freshman	22
4	Pamela	Alexander	Junior	21
5	Zonia	Kittleson	Junior	22
6	James	Smallwood	Senior	19
7	Heather	Beasley	Sophomore	18

Showing 1 to 100 of 100 entries

Figure 1: DB*Snap*'s User Interface.

Section 4 presents the comparison of DB*Snap* with other systems and tools to teach relational algebra queries. Section 5 presents a discussion of various facets of integrating DB*Snap* into the Database curricula. Section 6 concludes the paper.

2. DBSnap'S DESIGN

Fig. 1 shows DB*Snap*'s user interface. The main components in the window are: (1) Operator palette, (2) dataset palette, (3) query area, (4) relational algebra panel, (5) query result panel, and (6) node result panel. Building a query tree is highly intuitive and consists of dragging operator and dataset blocks and connecting them in the query area. As the user builds a new query, the query result panel is automatically updated with the current query result and the relational algebra panel with the corresponding relational algebra expression. The user can also inspect the intermediate result produced by any intermediate node by clicking on that node. The intermediate result will appear in the node result panel. The remaining part of this section describes in detail several components of DB*Snap*.

2.1 Dataset Palette

DB*Snap* includes by default a small database (University Database) composed of several relations (datasets or tables). This database has the following schema:

```
Students (SID, LName, FName, Level)      [100]
Courses (CID, CName)                     [20]
Professors (PID, LName, FName)           [20]
Course_Student (CID, SID)                [125]
Course_Professor (CID, PID)              [20]
```

The underlined attributes are the primary keys, i.e., attributes that uniquely identify a record in the dataset. The number included at the end of each dataset represents the number of records in this dataset. The University Dataset has a size and complexity that enables building simple and relatively complex queries while maintaining small query results that can be easily inspected. DB*Snap*'s dataset palette lists all the available dataset blocks and also includes a link to import additional datasets. This last feature enables customizing the tool for specific class assignments or projects. Each dataset block has a distinctive orange color, a left circular handle to connect the dataset with its parent node, and a right text area that shows the relation name. Observe in Fig. 1 that the block shape clearly shows that a dataset is a leaf node, i.e., it does not have a connection link to add a node underneath it.

2.2 Operator Palette

DB*Snap* supports a wide array of relational algebra operators including basic operators (e.g., Selection and Projection), Join operators, and highly useful extensions like the Aggregation operator. Each operator is represented as a block in the operator palette and each operator type has a distinguishing color. Fig. 2 shows a subset of the supported operators. In each sub-figure, the left tree is the query representation in DB*Snap* while the right one is the usual representation in database textbooks, e.g., [4, 12]. Observe that

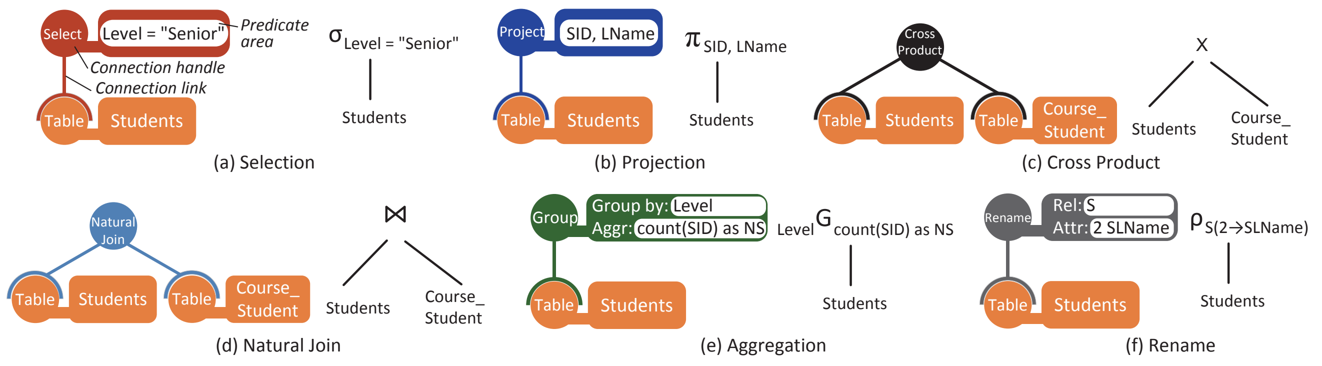


Figure 2: Some DB*Snap* Operators.

DB*Snap* queries closely follow the intuitive tree representations used in textbooks. A key property of DB*Snap* is the use of innovative block shapes aimed to facilitate block manipulation and query tree construction. As shown in Fig. 2.a, a DB*Snap* operator has in general three visual components: (i) top-left: a circular connection handle to connect the operator with a parent node, (ii) top-right: a predicate area to specify required operator information, and (iii) bottom: one or two connection links to connect this operator with its operand(s). The visual shape of operators make it easy to detect operators without required operands or predicates, and do not permit adding more operands than needed. We present next the details of several DB*Snap* operators.

- Selection: $\sigma_\theta(R)$. This operator selects all the records of relation R that satisfy the predicate θ. Fig. 2.a shows the query $\sigma_{Level=\text{“}Senior\text{”}}(Student)$. Observe that the predicate area is used to specify the selection condition ($Level$ = “*Senior*”).

- Projection: $\pi_{a_1,...,a_n}(R)$. This operator removes all the attributes of R not contained in $a_1, ..., a_n$. An example of this operator ($\pi_{SID,LName}(Students)$) is shown in Fig. 2.b. The predicate area in this case stores the list of attributes ($SID, LName$).

- Cross Product: $R \times S$. This binary operator pairs each record of R with each record of S. Fig. 2.c shows the graphical representation of *Students* $\times$ *Course_Student*.

- Theta-join (θ-join): $R \bowtie_\theta S$. Returns a similar result as the Cross Product but selecting only the rows that satisfy the predicate θ.

- Natural Join: $R \bowtie S$. This operator is similar to the θ-join where the θ predicate is the equality of all the common attributes between R and S. Fig. 2.d represents *Students* $\bowtie$ *Course_Student*. The implicit join predicate is *Student.SID=Course_Student.SID*.

- Aggregation: ${}_{g_1,...,g_m}G_{f_1(a_1),...,f_k(a_k)}(R)$. This operator groups the records of R forming a group for each unique occurring permutation of the grouping attributes $g_1, ..., g_m$. For each group, the operator computes the aggregation functions $f_1(a_1), ..., f_k(a_k)$ where $a_1, ..., a_k$ are attributes of R and the supported functions are *sum*, *count*, *average*, *maximum* and *minimum*. By default, $count(SID)$ counts all the occurrences of *SID* including duplicates. DB*Snap* also supports $distinct-count(SID)$ which counts only distinct values. DB*Snap* supports the use of $*$ instead of an attribute name. While $count(SID)$ ignores *null* values, $count(*)$ counts these values too. For convenience, DB*Snap* allows renaming the attribute corresponding to an aggregation function using the keyword “as”. Fig. 2.e shows the query ${}_{Level}G_{count(SID)}(Students)$ where the attribute containing the results of the aggregation function *count* has been renamed to *NS*. In this case, the predicate area has two fields: the top one stores the grouping attributes ($Level$) and the bottom one the aggregation functions ($count(SID)$).

- Other set operations. DB*Snap* also supports common set operations such as Set Union ($R \cup S$), Set Difference ($R - S$) and Set Intersection ($R \cap S$).

- Rename: $\rho_{S(i_1 \to b_1,...,i_k \to b_k)}(R)$. This operator changes the name of relation R to S and the name of the attribute at position i_j to b_j. Fig. 2.f represents the query $\rho_{S(2\to SLName)}(Student)$. The top predicate field stores the new relation name (S) and the bottom one specifies the position and new attribute name (2 $SLName$).

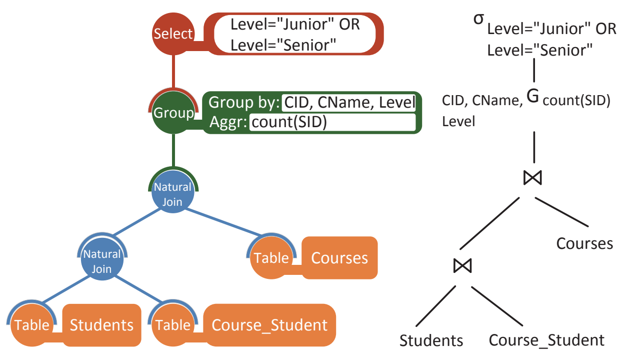


Figure 3: A DB*Snap* Query.

2.3 Query Area

The query area is where queries are constructed. This area gets dynamically resized as the query grows. Relational algebra queries in DB*Snap* are represented as trees. Representing a query as a tree is a powerful analogy to teach relational algebra and SQL (Structured Query Language) query plans. A query tree clearly shows the structure of a query and how the intermediate data generated by an operator is used as the input of other operators. In many database systems, in fact, a query (e.g., an SQL query) is parsed and converted into a query plan tree by the query processing engine. This plan can also be transformed into more efficient plans by the query optimizer using a set of transformation rules. Since DB*Snap* allows the dynamic manipulation of query trees, it is also an effective tool to teach query optimization.

Fig. 3 shows a query that computes the number of students of each level registered in each course considering only Junior and Senior students. The left tree is the DB*Snap* query while the right one is the equivalent query using the tree representation commonly found in textbooks. The DB-*Snap* query structure is very similar to the one of the conventional query tree, and the use of colors further simplifies the quick identification of operations involved in the query. In this example, it is easy to recognize that the query is combining three datasets, grouping the intermediate result and then selecting a subset of the groups. The relational algebra expression of this query is:

$$\begin{aligned}&\sigma_{Level=\text{``}Junior\text{''}\ \mathrm{OR}\ Level=\text{``}Senior\text{''}}\\&({}_{CID,CName,Level}G_{count(SID)}\\&((Students \bowtie Course\_Student)\\&\ \bowtie Courses))\end{aligned}$$

Even in this small example, the relational algebra expression may be intimidating for many students learning this language. DB*Snap* aims to simplify the understanding of queries by using an intuitive tree-based representation. Anytime the student adds a new block to the query, the system automatically shows the corresponding relational algebra expression. This feature is aimed to help students make the connection between a relational algebra expression and its intuitive tree-based representation.

2.4 Result Panels

As shown in Fig. 1, the query result panel, and the node result panel are located at the right-hand side of DB*Snap*'s user interface. The query result panel shows the results of evaluating the current query and is dynamically updated anytime the user modifies the query. This key feature allows students to see the query results as they build the query and also enables them to quickly explore the effects of query modifications. Many times, specifically while building complex queries, students need to examine the results of intermediate nodes. DB*Snap* supports exploring the results of any query node. When the student clicks on a query node, the intermediate results of this node are shown in the node result panel.

3. IMPLEMENTING DBSnap

One of the core goals while implementing DB*Snap* was to maximize its availability so that any student or instructor can use it without the need to use specialized software or hardware. With this goal in mind, we implemented DB*Snap* as a web application that only uses standard internet browser features (HTML5 and JavaScript). DB*Snap* can be used with most internet browsers (e.g., Internet Explorer, Firefox, Safari, and Chrome) and hardware devices (e.g., desktops, laptops, tablets, and smartphones).

Fig. 4 shows the architecture of DB*Snap*. As shown in this figure, the system has four main components: an HTML web page (DBSnap.html) and three JavaScript libraries (GUI.js, DBBlocks.js, and Morphic.js). DBSnap.html has three container elements: the DB*Snap* Query Canvas (which contains the block palettes and the query area), and the two result panels (which get populated with the query and current node results). DBSnap.html dynamically interacts with GUI.js to calculate the position and size of each HTML container, support the manipulation of blocks, and display query results. GUI.js, in turn, interacts with DBBlocks.js to support the manipulation of queries and query blocks. DBBlocks.js defines the visual representation and behavior of all the graphical elements in the Query Canvas: operator blocks, dataset blocks, palettes, and queries. DBBlocks.js includes the Query Handler module that maintains an internal representation of the current query. This module interacts with the Query Evaluation module to obtain the result of the current query and with the Query to Relational Algebra Translation module to obtain the relational algebra expression of the current query. DBBlocks.js uses Morphic.js, another JavaScript library. Morphic.js provides lower-level classes such as *Morph* which specifies basic behavior of blocks and *WorldMorph* which implements a canvas where other visual shapes (blocks) can be manipulated. Morphic.js was developed by Jens Mönig and is available under GNU license [10]. Morphic.js is also used in Snap! [11], a popular block-based programming application.

Another DB*Snap* implementation goal was to make it highly extensible. To this end, we have modularized the code and organized the class hierarchy such that it facilitates the addition of new operators.

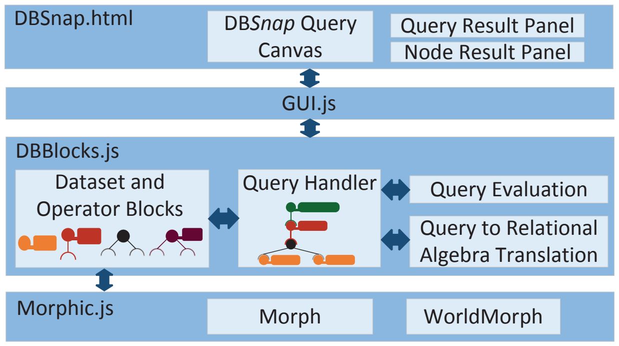


Figure 4: DB*Snap*'s Architecture.

4. RELATED LEARNING TOOLS

This section describes several related educational tools for database query languages, specifically for relational algebra. We describe the key properties of each tool and highlight important differences between these tools and DB*Snap*. Fig. 5 summarizes the key supported features of the considered tools.

WinRDBI. WinRDBI [3] is a Java-based educational application that allows the specification of queries in relational algebra, relational calculus and SQL. The interface provides a panel to write the query, another one to show the query results, and a database browser to explore the available relations. WinRDBI allows both specifying relations and importing them. The tool allows naming queries such that they can be used as operands in other queries. The key differences

Feature	**DB*Snap***	WinRDBI	iDFQL	Relational	RALT	Query Visualiser	Bags
Implementation technologies	HTML5, JavaScript	Java	Borland C++	Python	Java Swing	.NET, Mono	HTML5, JavaScript
Publicly available	●	●	●	●	●	●	●
Open source code	●			●		●	●
Block-based query editor	●		●		●		●
Shows RA expression	●	●		●		●	
Web application	●						●
Uses tree-based representation to build queries	●		●		●		
Automatically updates query result	●				●		
Intermediate results	●		●		●	●	●
Side-by-side queries	●	●	●		●	●	●
Can work without DB connection	●	●		●		●	●
Build-in datasets	●	●		●		●	●
Allows importing or connecting to custom data	●	●	●	●	●	●	
Views				●		●	

Figure 5: Comparison of Educational Tools for Relational Algebra Queries.

in comparison with DB*Snap* are that WinRDBI: (i) is not a block-based tool, (ii) requires the user to write text-based queries, e.g., requires writing a correct relational algebra expression, (iii) does not support the tree-based query representation, and (iv) does not automatically show the query results as the query is being built.

iDFQL. This tool provides a graphical interface to create relational algebra queries [1]. The tool allows creating a query using a set of icons or blocks that form a flow diagram. A key difference with DB*Snap*, is that operator blocks in iDFQL do not include the connection links, these links have to be added separately using a toolbar component. Operators in DB*Snap* already include the expected number of connection links simplifying the creation of a query. Furthermore, the predicates of an operator in iDFQL are represented using additional blocks while in DB*Snap* they are integral part of the operator block. When a query is built, iDFQL does not show the actual relational algebra expression (limiting its usefulness as a tool to learn relational algebra) and does not automatically show the query results. Moreover, the tool requires a connection with an external database, is not currently maintained, and suffers from some software compatibility issues, e.g., did not run in Windows 7 or 8.

Relational. This educational tool implemented in Python provides a workspace to write relational algebra expressions [14]. The tool allows the visualization of results and the exploration of available relations. Queries in Relational are text-based and specified in a single line. Operators are represented as special Unicode symbols that can be inserted using the buttons of a toolbar. Relational is not a block-based tool, does not support tree-based queries, does not dynamically show the query results, and does not allow the exploration of intermediate results.

RALT. This tool allows building relational algebra queries using an extended query tree structure [9]. Like DB*Snap*, RALT represents queries using trees. RALT, however, adds intermediate nodes (tables) to show the results of every operator and uses additional nodes to represent operator predicates. The inclusion of these extra nodes significantly increases the size of the query and it quickly fills the available query space. Constructing queries is simple for very small queries but becomes complex for relatively large queries. RALT requires a connection with an external database and does not show the relational algebra expressions of queries.

Query Visualiser. Query Visualiser (QV) [2] is another tool that allows the specification of written relational algebra expressions. After a query is built, QV generates a hierarchical representation of the query that enables exploring its intermediate results. QV also supports side-by-side query comparison and reports various query evaluation statistical measures. The key difference with DB*Snap* is that QV doesn't use a block-based approach. Instead, QV requires users to specify text-based queries. Another difference is that QV does not automatically show the query results while a query is modified.

Bags. Bags is a block-based tool to build relational algebra queries [5]. Like DB*Snap*, Bags allows the user to build a query by dragging and connecting blocks. The tool allows the visualization of the query results and includes several built-in datasets. A fundamental difference with DB*Snap*, however, is the type of query representation used by the tool. Bags was built extending Snap! [11], a block-based tool designed to teach conventional computer programming. A program in Snap! looks, in fact, very similar to the corresponding code in an imperative programming language. The operators and relation blocks of Bags have exactly the same look and feel as blocks in Snap!. A query in Bags is not a relational algebra expression nor a direct representation of the corresponding query tree. Moreover, the tool does not show the corresponding relational algebra expression of a query. DB*Snap* in contrast uses a tree-based query representation that has been extensively used by database educators and textbooks as an intuitive way to represent a relational algebra query. Additionally, DB*Snap* shows the relational algebra expression of any query specified in the tool. Another difference is that a block in Bags does not prevent the inclusion of a sequence of two or more consecutive blocks as a single operand (which would produce an incorrect query). In the case of DB*Snap* the rounded shape of a connection link allows the inclusion of only one block (or rooted sub-query). Bags allows the exploration of the result of any query node but does not automatically update the overall query result as the user builds the query.

While DB*Snap* is not the first tool to build relational algebra queries, it includes a set of features aimed to make it highly effective to teach relational algebra and database queries in general. Specifically, DB*Snap* differentiates from the rest in that it is the only block-based tool that uses a tree-based query representation that fully matches the highly intuitive query trees used by many educators and textbooks. Furthermore DB*Snap* is a highly interactive tool that automatically shows the query results and the equivalent relational algebra expression while a query is being modified, and allows the inspection of the output of any intermediate node.

5. DISCUSSION

DB*Snap* can be used to teach relational algebra and query optimization in introductory and advanced database courses. DB*Snap* is a publicly available web application [13] that can be used by educators and students on a wide array of devices. DB*Snap*'s extended availability makes it a suitable tool to be used in in-class activities as well as in assignments and projects. In the case of relational algebra, a common learning objective is that students understand the semantics of key operators and are able to build queries to answer specific questions. DB*Snap* can be an important resource to meet this goal. The tool can be used by the instructor to introduce and show the output of each operator and to highlight the relationships between query trees (also used in database textbooks) and the corresponding relational algebra expressions. DB*Snap* can then be used in hands-on class activities where students explore the operators and build several assigned queries. The interactive features of DB*Snap* (automatically showing the query result and corresponding relational algebra expression) make it an environment where students can experiment with different query arrangements, get immediate feedback, and improve their solutions until they find the correct answers. Instructors can also prepare projects or assignments where students use DB*Snap* to build queries that generate specific reports. Since DB*Snap* supports importing datasets, instructors can prepare custom datasets for these assignments. DB*Snap* can also be used to teach database query optimization. Specifically, DB*Snap* can be used to show that multiple query arrangements (generated by query transformation rules) can produce the same result and that some arrangements may be faster than others. The use of query trees (query plans) is in fact one of the most useful representations of a query to teach the effects of transformation rules. DB*Snap* can be used to dynamically transform a query into an equivalent one by re-arranging the query blocks.

To assess the effectiveness of DB*Snap* as a learning tool, the authors used it in a relational algebra class and prepared an end-of-class survey. While this was a small class, the results are promising. The survey results are presented next (1: Agree - 4: Disagree). To the question "*I am able to recognize the key properties of relational algebra operators and build queries to answer specific questions*", 80% of students agree and 20% moderately agree. To the question "*The use of DBSnap helped me to have a clear understanding of relational algebra operators*", 80% of students agree and 20% moderately agree. To the question "*DBSnap significantly helped me to learn how to build relational algebra queries*", 78% of students strongly agree and 22% moderately agree. In the open ended questions, students gave positive comments about DB*Snap*, e.g., "*DBSnap really helps the user follow the query as it is being built*", "*It is very easy to make changes on the query by rearranging blocks and subtrees*", "*It is so simple, user-friendly and I liked all the colors*", and "*The ability to see the result of every node is very helpful*".

6. CONCLUSIONS

A clear understanding of relational algebra operators and the ability to write relational algebra queries are two key learning objectives in database courses. Furthermore, a solid grasp of relational algebra and query plans provides a good foundation to learn the SQL-like query languages that are used in many modern database systems. This paper introduces DB*Snap*, an open-source tool aimed to facilitate the learning of relational algebra and query plans. One of DB*Snap*'s key and differentiating features is that it uses a tree-based query structure that is highly similar to the intuitive query trees used by many textbooks and educators. DB*Snap* is also a highly interactive tool that shows the query result and equivalent relational algebra expression while a query is being built. This paper presents the different design features of DB*Snap*, provides its implementation details, describes guidelines to use it in database courses, and evaluates is effectiveness as an educational tool.

7. REFERENCES

[1] A. P. Appel, E. Q. Silva, C. Traina, and A. J. M. Traina. idfql: A query-based tool to help the teaching process of the relational algebra. In *WCETE*, 2004.

[2] G. Constantinou. Relational algebra and sql query visualisation. Technical report, Dept. of Computing, Imperial College, 2010.

[3] S. W. Dietrich, E. Eckert, and K. Piscator. Winrdbi: A windows-based relational database educational tool. In *ACM SIGCSE*, 1997.

[4] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 6 edition, 2010.

[5] J. Gorman, S. Gsell, and C. Mayfield. Learning relational algebra by snapping blocks. In *ACM SIGCSE*, 2014.

[6] S. H. Kim and J. W. Jeon. Programming lego mindstorms nxt with visual programming. In *ICCAS*, 2007.

[7] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: Urban youth learning programming with scratch. In *ACM SIGCSE*, 2008.

[8] A. Marron, G. Weiss, and G. Wiener. A decentralized approach for programming interactive applications with javascript and blockly. In *AGERE!*, 2012.

[9] P. Mitra. Relational algebra learning tool. Technical report, Dept. of Computing, Imperial College, 2009.

[10] J. Mönig. morphic.js - source code. `https://github.com/jmoenig/morphic.js`.

[11] C. North and B. Shneiderman. Snap-together visualization: Can users construct and operate coordinated visualizations? *Int. J. Hum.-Comput. Stud.*, 53(5):715–739, 2000.

[12] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., 6 edition, 2010.

[13] Y. N. Silva and J. Chon. Dbsnap. `http://www.public.asu.edu/~ynsilva/dbsnap`.

[14] S. Tomaselli. Relational - educational tool for relational algebra. `https://github.com/ltworf/relational/`.

[15] D. Wolber. App inventor and real-world motivation. In *ACM SIGCSE*, 2011.