Querying Databases by Snapping Blocks

Yasin N. Silva
Arizona State University
Glendale, AZ 85306, USA
Email: ysilva@asu.edu

Jaime Chon
Arizona State University
Glendale, AZ 85306, USA
Email: jchon@asu.edu

***Abstract*—A key area of focus in recent Computer Science education research has been block-based programming. In this approach, program instructions are represented as visual blocks with shapes that control the way multiple instructions can be combined. Since programs are created by dragging and connecting blocks, the focus is on the program's logic rather than its syntax. In this demonstration we present DB*Snap*, a system that enables building database queries, specifically relational algebra queries, by connecting blocks. A differentiating property of DB*Snap* is that it uses a visual tree-based structure to represent queries. This structure is, in fact, very similar to the intuitive query trees commonly used by database practitioners and educators. DB*Snap* is also highly dynamic, it shows the query result and the corresponding relational algebra expression as the query is built and enables the inspection of intermediate query results. This paper describes DB*Snap*'s main design elements, its architecture and implementation guidelines, and the interactive demonstration scenarios. DB*Snap* is a publicly available system and aims to have a transformational effect on database learning.**

I. Introduction

The creation of a computer program requires a logical specification of the instructions to be executed and a fully correct syntactical representation of the instructions. This second component is a common reason of frustration for programmers, particularly those that are beginners, because minor syntactical errors prevent the execution of logically correct programs. Recognizing this limitation of common programming environments, the research community on computer science education has recently focused on the design and study of block-based programming environments such as Snap! [1], Scratch [2], Blockly [3], Mindstorm [4], and App Inventor [5]. In this approach, computer programs are created by dragging and connecting blocks and consequently the focus is on the program's structure and logic instead of its syntax. Block-based systems have revolutionized the way computer programming can be taught and have enabled younger students to learn fundamental programming concepts.

In this demonstration we present DB*Snap*, a web application to build database queries, particularly relational algebra queries, by snapping blocks. An important feature of DB*Snap* is that it uses a tree-based structure to visually represent a query. Tree-based query representation has been extensively used by database educators and textbooks because it intuitively shows the organization of the different operators and the way intermediate results flow in the query pipeline. DB*Snap* also dynamically shows the query result as the query is being constructed and allows the exploration of intermediate node results. This paper presents the design and implementation details of DB*Snap*, which are aimed to enable other researchers to extend and customize DB*Snap*. The study and evaluation of DB*Snap* as an educational tool was recently presented in [6]. DB*Snap* is a publicly available tool and aims to have the same transformational effect on database learning as other block-based systems had on traditional programming learning.

The rest of the paper is organized as follows. Section II presents the design elements of DB*Snap*. Section III describes the architecture and implementation details of DB*Snap*. Section IV describes the demonstration scenarios and Section V concludes the paper.

II. DBSnap's Design

As shown in Fig. 1, the main interface components of DB*Snap* are: (1) operator palette, (2) dataset palette, (3) query area, (4) relational algebra panel, (5) query result panel, and (6) node result panel. To build a query, the user only needs to drag operator and dataset blocks and connect them in the query area. As the user builds a query, the query result panel and the relational algebra panel are automatically updated with the query result and the corresponding relational algebra expression, respectively. When the user clicks on any node, the result of this node appears in the node result panel. This feature enables the inspection of intermediate results. We describe next the different components of DB*Snap*.

A. Dataset Palette

The dataset palette shows the list of all the available dataset blocks (relations or tables). DB*Snap* includes an initial database (University Database) with the following schema:

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

The number at the end of each relation represents the number of records in that relation. The sample University Database has a size and complexity that allow building relatively complex queries while maintaining small query results that can be easily visualized. Moreover, DB*Snap* allows importing additional datasets. This can be done by clicking on the *Import Data Set* link at the bottom of the dataset palette. Each dataset block is a terminal (leaf) node and thus its graphical representation does not allow connecting blocks underneath it. Dataset blocks have a distinguishable orange color, a left circular handle to connect the dataset with its parent node, and a right text area

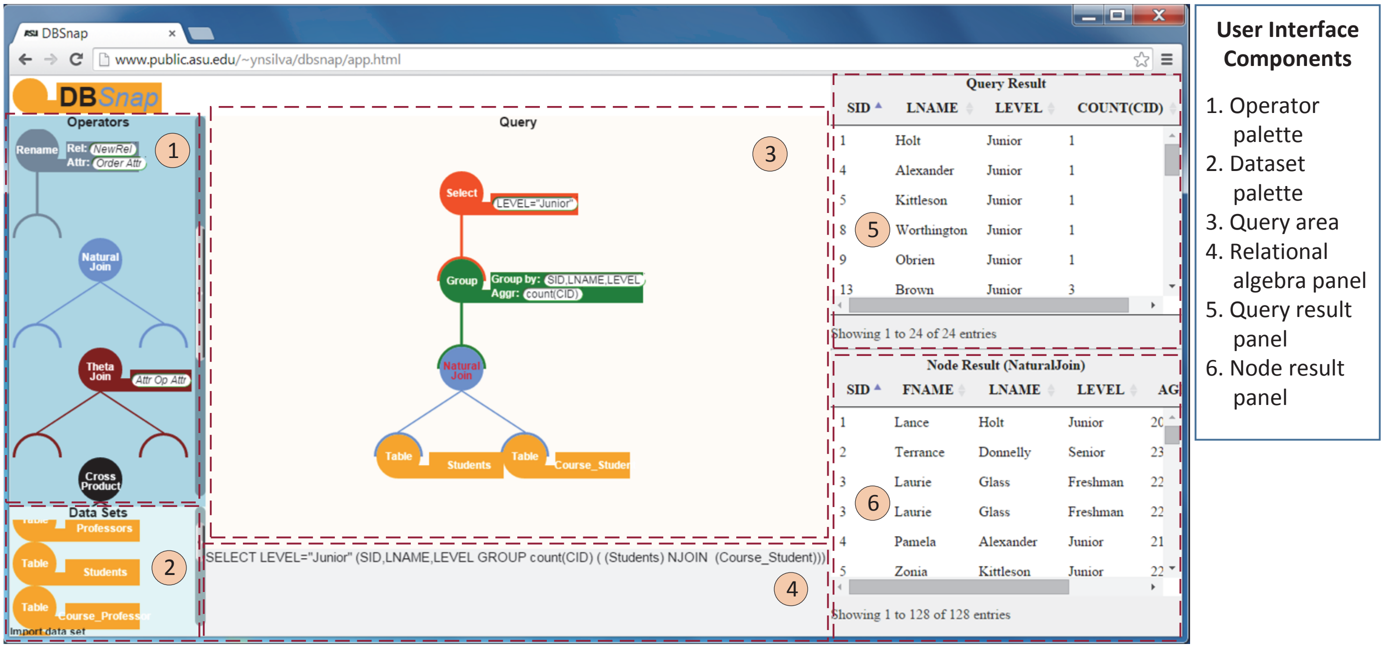

Fig. 1. DB*Snap*'s User Interface.

with the name of the relation. The bottom part of the DB*Snap* query in Fig. 2.a shows the dataset block Students.

B. Operator Palette

The operator palette contains the set of available operator blocks. Each supported relational operator is represented as an operator block with a distinguishing color. As presented in Fig. 2.a, each operator block has, in general, three visual components: (1) top-left: a circular connection handle to connect the operator with a parent node, (2) top-right: a predicate area to specify the required operator parameters, and (3) bottom: one or two connection links to connect this operator with its operand(s). The shape of DB*Snap* operator blocks facilitate block manipulation and query tree construction. Particularly, the shape of an operator makes it easy to identify missing predicates and children nodes, and does not allow assigning more operands than needed. Fig. 2 shows two of the supported operators. In each sub-figure, the left tree is the DB*Snap* query and the right one is the query tree representation commonly used by database practitioners and in database textbooks, e.g., [7], [8]. Observe that DB*Snap* queries closely follow the intuitive query trees used by database professionals and educators.

DB*Snap* supports many relational algebra operators including basic operators (e.g., Selection and Projection), set-based operations (i.e., Union, Intersection, Difference and Cross Product), Join operators, and useful extensions like the Grouping operator. We describe next 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 use of this operator ($\sigma_{Age>21}(Student)$). Observe that the predicate area is used to specify the selection condition ($Age > 21$).

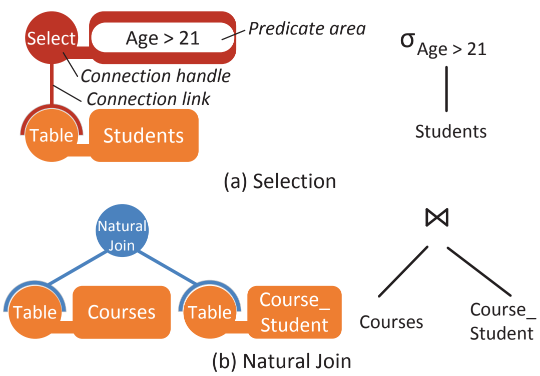

Fig. 2. Some DB*Snap* Operators.

- Projection: $\pi_{a_1,\ldots,a_n}(R)$. This operator removes all the attributes of R not contained in $a_1, \ldots, a_n$. The predicate area in this case stores the list of attributes $(a_1, \ldots, a_n)$.
- Cross Product: $R \times S$. This operator pairs each record of R with each record of S. Since this is a binary operator, it has two connection links.
- 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 predicate θ is the equality of all the common attributes between R and S. Fig. 2.b represents *Courses* $\bowtie$ *Course_Student*. The implicit join predicate is *Courses.CID=Course_Student.CID*.
- Grouping: ${}_{g_1,\ldots,g_m}G_{f_1(a_1),\ldots,f_k(a_k)}(R)$. This operator groups the records of R forming a group for each unique occurring permutation of the grouping

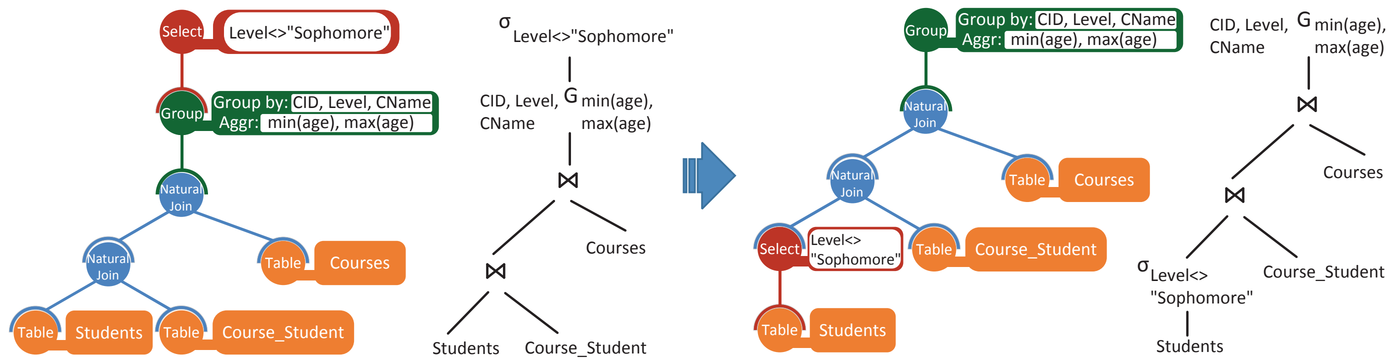

Fig. 3. DB*Snap* Queries.

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*, *avg*, *max* and *min*. 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". The predicate area of this operator has two fields: the top one stores the grouping attributes and the bottom one the aggregation functions.

- Other set operations. DB*Snap* also supports common set operations such as Set Union ($R \cup S$), Set Intersection ($R \cap S$) and Set Difference ($R - S$).
- 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. The top predicate field of this operator stores the new relation name and the bottom one specifies the positions and new attribute names. Alternatively, the operator supports the direct specification of the old attribute names instead of their positions.

C. Query Area

The query area is the component where queries can be created. This area expands as the query grows. A DB*Snap* query is specified as a tree of connected dataset and operator blocks. Representing a query as a tree is a very useful analogy because it closely represents the way the data is processed by the different operators and how the results of intermediate operations are used as the input of other operators.

Fig. 3 shows two DB*Snap* queries. Next to each query, the figure includes the common query tree representation used by database practitioners. The left DB*Snap* query computes, for each course and level, the age range of enrolled students. The query ignores the Sophomore level. The tree-based query structure and the use of distinctive block colors make it easy to understand the semantics of a DB*Snap* query. In this query for instance, it is easy to recognize that it joins three datasets, groups the intermediate result and then filters some of the groups. The relational algebra expression of this query is:

$$\sigma_{Level<>\text{"Sophomore"}} \\ ({}_{CID,Level,CName}G_{min(age),max(age)} \\ ((\mathit{Students} \bowtie \mathit{Course\_Student}) \bowtie \mathit{Courses}))$$

Even in this small query, the relational algebra expression may be intimidating for non-expert users. DB*Snap* aims to simplify the process of building a query by using an intuitive query structure and by showing the corresponding relational algebra expression after any change in the query.

An important feature of DB*Snap* is that it allows rapid query modification. This feature makes DB*Snap* an excellent tool to learn about query transformations and query optimization. Many database systems transform the initial query plan into several alternative plans using query transformation rules. The query optimizer selects the query with the lowest estimated cost for execution. For instance, the right query of Fig. 3 shows a query that is equivalent to the left one. In this case, the selection operator has been pushed below the grouping and join operators. Since the right query executes the selection earlier, it reduces the number of tuples to be joined and aggregated and has a potentially smaller cost. Using DB*Snap* a user can quickly transform the left query into the right one. Furthermore, DB*Snap* supports the side-by-side specification of both queries.

D. Result Panels

The query result panel and node result panel are located on the right-hand side of DB*Snap*'s user interface (see Fig. 1). The query result panel shows the result of the current query. This panel gets automatically updated every time the user makes a change in the query, i.e., adding, removing or updating an operator. This feature helps the user to refine a query or explore the effects of certain changes. The node result panel shows the result of any selected node (which is also highlighted in the query area). This feature is particularly useful to explore the data generated by intermediate nodes in complex queries. Both result panels also allow sorting their content by any attribute.

III. DBSnap Architecture and Implementation

Two important goals of DB*Snap*'s implementation were to maximize its availability and to facilitate its extensibility.

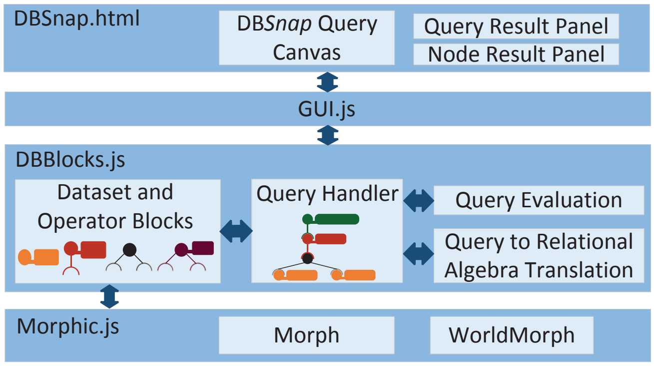

Fig. 4. DB*Snap*'s Architecture.

To achieve the first goal, we implemented DB*Snap* as a web application using only standard internet browser features, particularly HTML5 and JavaScript. DB*Snap* does not require any specialized software or hardware. In fact, DB*Snap* can be used with most internet browsers (e.g., Chrome, Firefox, Internet Explorer, and Safari) and hardware devices (e.g., desktops, laptops, tablets, and smartphones). To achieve the second goal and facilitate the addition of new operators, we modularized the code following the Model-View-Controller design pattern (MVC).

Fig. 4 presents DB*Snap*'s architecture. 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 (View) has three container objects: the DB*Snap* Query Canvas, which contains the two 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 (Controller) to calculate the position and size of the HTML containers, support the manipulation of blocks in the Query Canvas, and display the query results. DBBlocks.js (Model) maintains the internal representation of the application (palettes, queries, and blocks). This component includes the Query Handler module which maintains an internal tree-based representation of the current query. The Query Handler module interacts with the Query Evaluation module to evaluate the current query and with the Query to Relational Algebra Translation module to generate the relational algebra expression of the current query. DBBlocks.js is built on top of the Morphic.js framework, a JavaScript library developed by Jens Mönig and available under the GNU license [9]. Morphic.js provides lower-level classes to handle user input and redrawing of dirty frames, and enables basic functionality like dragging, dropping, and connecting blocks. Morphic.js is also used in Snap! [1], a well known block-based programming application.

IV. Demonstration Scenarios

DB*Snap* is a publicly available web application [10] that can be used on a wide array of devices. The demonstration of DB*Snap* is aimed to be highly dynamic and will be composed of two parts. In the first part, we will briefly explain the design and architecture of the application. We will also introduce the various operators and show several sample queries using the University Database presented in Section II. The second part will be focused on providing a hands-on experience with DB*Snap*. We will bring several devices (laptops and tablets) so that conference attendees can use DB*Snap* to build various queries. Attendees will be able to use the built-in University Database and construct queries like the ones presented in Fig. 3. They will also be able to experiment with various features of DB*Snap* like the automatic generation of relational algebra expressions and the inspection of intermediate results. To show the use of DB*Snap*'s feature to import custom datasets, we will prepare files containing the tables of the TPC-H database benchmark [11]. After importing these tables, attendees will be invited to build some of the TPC-H queries.

Given that one of the goals of building DB*Snap* was to create a highly intuitive tool to learn database query languages, we plan to engage in discussion with other database educators about ways in which the use of DB*Snap* can be integrated into database courses.

V. Conclusions

This paper describes DB*Snap*, an interactive web application that enables building database query trees by snapping blocks together. A key feature of DB*Snap* is that it uses a tree-based representation of queries. This representation is very similar to the query trees commonly used by database practitioners and educators. DB*Snap* is also a highly interactive tool that shows the query result and the relational algebra expression while the query is constructed. This paper describes the design features of DB*Snap*, presents its architecture and key implementation details, and describes the demonstration scenarios.

While DB*Snap* was originally built as an educational tool, its approach to constructing queries by dragging and connecting blocks to form query trees can also be a more user-friendly alternative to specify queries in many real-world systems. DB*Snap*, in fact, could be an alternative to other graphical query languages, e.g., Query by Example (QBE).

References

[1] C. North and B. Shneiderman, "Snap-together visualization: Can users construct and operate coordinated visualizations?" *Int. J. Hum.-Comput. Stud.*, vol. 53, no. 5, pp. 715–739, 2000.

[2] 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.

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

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

[5] D. Wolber, "App inventor and real-world motivation," in *ACM SIGCSE*, 2011.

[6] Y. N. Silva and J. Chon, "Dbsnap: Learning database queries by snapping blocks," in *ACM SIGCSE*, 2015.

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

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

[9] J. Mönig, "morphic.js - source code," https://github.com/jmoenig/morphic.js.

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

[11] "Tpc-h version 2.17.0," http://www.tpc.org/tpch.