The Interactive Join: Recognizing Gestures for Database Queries

Arnab Nandi
Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
arnab@cse.osu.edu

Michael I. Mandel
Computer Science and Engineering
The Ohio State University
Columbus, OH, USA
mandelm@cse.osu.edu

CHI'13, April 27 – May 2, 2013, Paris, France.
ACM 978-1-XXXX-XXXX-X/XX/XX.

Abstract
Direct, ad-hoc interaction with databases has typically been performed over console-oriented conversational interfaces using query languages such as SQL. With the rise in popularity of gestural user interfaces and computing devices that use gestures as their exclusive mode of interaction, database query interfaces require a fundamental rethinking to work without keyboards. Unlike domain-specific applications, the scope of possible actions is significantly larger if not infinite. Thus, the recognition of gestures and their consequent queries is a challenge. We present a novel gesture recognition system that uses both the interaction and the state of the database to classify gestural input into relational database queries. Preliminary results show that using this approach allows for fast, efficient and interactive gesture-based querying over relational databases.

Author Keywords
Gesture Recognition, Query Interface, Databases, Unsupervised Classification

ACM Classification Keywords
H.5.2 [Information interfaces and presentation: User Interfaces – Graphical user interfaces.]:

Introduction

Gestural user interfaces have become a popular mode of interaction with a wide variety of touch-based, motion-tracking or eye-tracking devices. Given the rising popularity of such devices, domain-specific applications have come up with mappings between standard gestures and actions pertinent to the system. The onus of gesture recognition is on the user interface layer, which identifies the gesture as one of a set of gestures predefined by the operating system. The gesture type, with parameters such as *coordinates* and *pressure* are sent to the application, which then uses them to infer actions. This mapping of gestures to actions can be considered as a **classification problem**, and the bulk of the recognition is performed at the interface layer, independent of the application state.

End-user-friendly interaction with databases is a well-motivated problem [6]. There has been a wide variety of work in open-domain query interfaces, however all current efforts are based on keyboard or mouse-driven interaction, and are hence unsuitable for gestures.

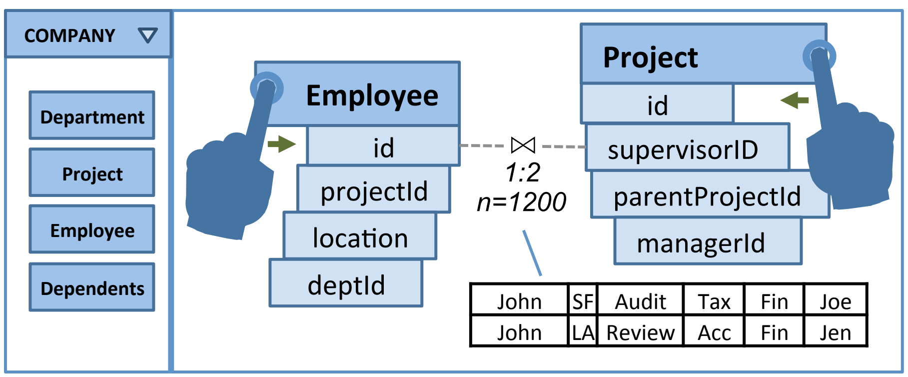

Figure 1: The Interactive Join being performed in the QWiK interface. Tables in a relational database are joined together using multitouch gestures. As two tables are brought close to each other, the attributes are presented in an arc such that they are amenable to be joined. The most likely join is presented as a preview to the user.

In the context of ad-hoc, open-domain querying of relational databases, the use of gestures as the sole mode of interaction faces several challenges. First, the space of possible actions is large[1] – the action depends on the underlying database query language (e.g., SQL), the schema of the database (the tables and the attributes for each table) and the data contained in it (the unique values for each attribute, and the individual tuples for each table). One solution to this problem is to present a modal interface allowing the user to first pick the type of query, and the drill down the parameters of the desired query. Such a modal interface goes against the desiderata of a fluid, directly manipulable interface.

Thus, there is a strong need for the system to prioritize the most likely query. This prioritization is performed in a principled and intelligent manner by our novel two-stage classifier, which relies on both gestural input and the state of the database to recognize queries. At the first level, the classifier identifies an ordering of candidate query types that are most likely to be associated with the gesture. At the second level, the classifier leverages various metadata stored in the database such as the schema, type information and data distributions for each attribute to narrow down on the exact query being formulated. In addition, the classifier needs to be either unsupervised or trained *offline* to avoid custom training on a per-user level. And it must work at *interactive* speeds, such that it updates the query selected by the classifier as the gesture evolves. This allows the user to gain insight into the database, and at the same time disambiguate the gesture, improving gesture recognition.

[1] When considering n-ary joins, this space can be infinite.

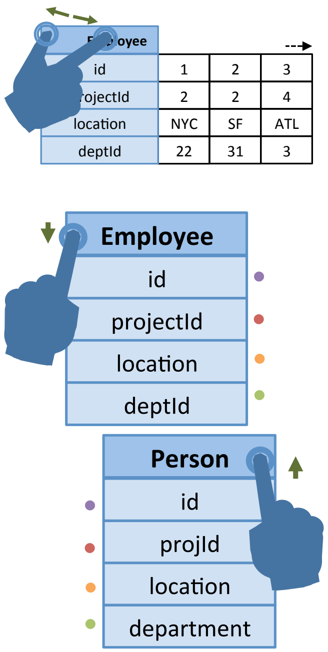

Figure 2: The PREVIEW and UNION actions. Colored dots represent corresponding fields in the UNION.

A Gesture-driven Database Query Interface

As depicted in Figure 1, the database query interface allows users to directly manipulate results by interacting with them in a sequence of gestures. Tables are dragged to the workspace from the database tray. Each table in the workspace represents a *view* of the table, i.e. cloned representation of the data, and can be directly manipulated. Each *gesture* denotes a single manipulation *action* and impacts only the cloned instance – not the original database. There are a finite number of intuitive gestures that the user can learn, each of which when performed on the workspace can correspond to an action. Users can undo each action to return to the previous workspace state. Since actions directly correspond to SQL queries, all actions manipulate the target SQL view to another SQL view. Thus, actions are stackable and can be performed in sequence, manipulating tables in the workspace till the desired result is achieved.

A Gesture Vocabulary

While the size of our gesture vocabulary (and the set of corresponding types of actions) is quite large, we present only three actions for the sake of conciseness and clarity. The three actions, `UNION`, `JOIN` and `PREVIEW` allow the user to view, compose and combine information from the database. The gestures associated with these actions are shown in Figures 1 and 2. This condensed vocabulary allows us to motivate and demonstrate various aspects of the gesture recognition problem.

`PREVIEW`**:** This action works on a single table. When dragged from the database tray, each table is represented by the name of the table and its attributes. By making the *pinch-out* gesture on the table in the workspace, the `PREVIEW` action is issued on the target table. This is issued to the database as the SQL query `SELECT * FROM TARGET_TABLE LIMIT 10;`, presenting the first ten rows of the table on screen[2].

`JOIN`**:** Two tables can be composed together by moving them close to each other. The `JOIN` action represents the *inner equijoin* SQL query, representing combinations of rows that have the same value for the attributes that are being joined upon. Upon bringing tables close to each other, the attribute list curves such that users can articulate the intended pair of attributes. The design considerations for this curvature are described in Section "Design Considerations: Join Layout".

`UNION`**:** Two tables can be unified into a single table if their attributes are compatible; i.e. they have the same number of attributes, and each pair of attributes is of the same data type. To unify tables, the user drags one table onto another from the top, in a stacking gesture. A preview of the unified rows representing both originating tables is depicted.

Gesture Recognition as Classification

We assume the input to our classifier is similar to what is available on current multitouch mobile platforms. The UI layer will supply the classifier with a list of (x,y) coordinates and an ordinal identifier indicating which finger it is associated with. These identifiers are assigned arbitrarily when a gesture is initiated, but are consistent over the course of the gesture. Given this input, the classifier makes a decision based on the most recent coordinate for each identifier.

A gesture is classified as a particular query according to the **proximity** and **compatibility** of the tables involved. Proximity encompasses all of the spatial information

[2]We skip more complex variants of preview for conciseness and to focus on the gesture recognition challenges.

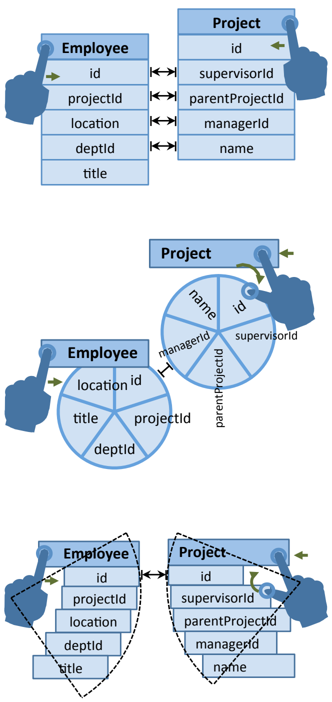

Figure 3: Choices of Table Layout: Tabular, Radial and Arc. The responsive arc layout allows all queries to be expressed unambiguously while still achieving readability.

about the UI elements, including size, shape, position, and orientation of table and field graphical representations along with their velocity and acceleration.

Classification based solely on proximity is the currently prevalent UI paradigm. By adding compatibility, we are able to increase the likelihood of selecting semantically meaningful queries. Compatibility criteria include schema information like field type, and data distributions, like histograms, extreme values, intersection in random samples, or total intersection.

We use a maximum entropy classifier in which we define many "features" of queries, including proximity and compatibility features conditioned on each type of query, and combine them linearly in the argument of an exponential. Mathematically, the goodness $g(q)$ of a potential query q with feature values $f_i(q)$ is

$$g(q) = \exp(\sum_i \lambda_i f_i(q)). \quad (1)$$

Features can be binary or real-valued, with 0 being the value of an uninformative feature. In our two-stage approach, features are computed in a specific order and earlier features that have a value of $-\infty$ stop the computation of later features, avoiding unnecessary calculations. Parameters of the classifier, λ_i, can be learned across a collection of recorded training gestures to tune the quality of the classifier. For the preliminary experiments in Section "Preliminary Results", these parameters are set manually. Learning these tunings from data will allow future quality improvements. New queries can be defined by adding new feature functions $f_i(q)$ and adding new potential queries q to the set of queries.

At each classification request, each query type (`JOIN`, `UNION`, `PREVIEW`) independently selects a specific query with the highest goodness and the query with the highest overall goodness is selected. Each query type first considers whether the current gesture could represent a query of its type, i.e., it involves the correct number of tables and the tables are compatible with each other and with the query type. If so, it proceeds to find the best specific query of its type, checking the proximity and compatibility of the fields and tables involved, if necessary.

Design Considerations: Join Layout

While the `PREVIEW` and `UNION` interactions are straightforward, laying out attributes during the `JOIN` interaction faces several challenges. First, due to the textual nature of the information, it needs to be presented such that readability is preserved. Second, the interface should allow the user to express all possible queries. In the case of a pair of tables with m and n attributes of the same type, there are $m \times n$ possible joins.

We consider multiple layout options to represent the `JOIN` operation, as shown in Figure 3. The first option is to present attributes as simple vertical lists. The problem with such a layout is that for any position of two vertical lists, many pairs of attributes can be at identical distances, leaving the `JOIN` intent ambiguous. For example, aligning a pair of attributes at the top of their respective tables' lists will always result in aligning the second attributes at the same distance.

A second option is to consider a layout where each table is represented as a radial menu. Geometrically, two circles are closest at exactly one location, uniquely specifying a pair of attributes. However, radial menus are hard to read, don't scale to a large number of attributes, and need to be rotated to allow all $m \times n$ attribute pairings.

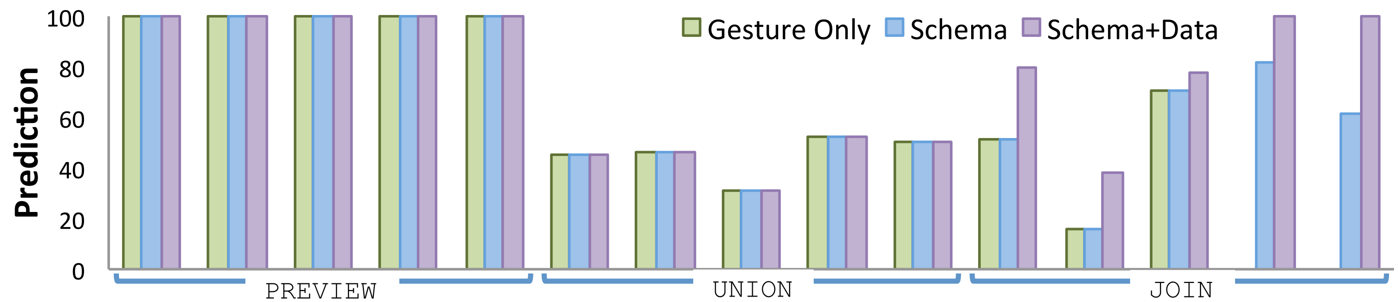

Figure 4: Prediction scores for a workload of 15 queries, where 100% represents correct classification at the start of the gesture articulation, and 0% represents an incorrect classification even after completion of the gesture. Information from the schema and data in the database allows our classifier to better predict the intended database query for ambiguous gestures.

As a solution to these problems, we use an *arc* layout, such that attributes are vertically stacked ensuring readability, but are placed in an arc connected at the table label, ensuring unambiguous joining intent. Further, making the orientation of the arcs flexible and user controllable (a multitouch interaction involving four points, two for each table) allows any pair of attributes to be selected as the join predicate.

Preliminary Results

We now share some preliminary results on the quality of our gesture recognition system. Experiments were performed by collecting multitouch coordinates (a series of X, Y coordinates per touched finger) from a prototype system implemented using Javascript and HTML on an iPad. Each set of coordinates represents a multitouch gesture on one or more tables. For `JOIN` queries, we test the classifier by not allowing for rotations to the arc, forcing the classifier to resolve ambiguity. The test database used the tables shown in Figure 1, each containing 4 attributes with varying types and data distributions. Coordinates were collected from a single user for 15 different gestures, articulating 5 each for `JOIN`, `UNION` and `PREVIEW`.

We identify three key metrics: **Accuracy**, **Prediction** and **Performance**. Accuracy measures the fraction of the queries that our classifier correctly identified from the gestures. Prediction measures how quickly our classifier can correctly map the gesture to the intended query. It is calculated as the fraction of touch coordinates necessary for the query to be correctly identified. Performance measures how quickly the system can react to a given input, measured as the number of milliseconds it takes to identify the query at each new touch coordinate.

For performance, we observed that our system performs well within the 10ms range per classification and can thus maintain a fluid touch interaction. As expected, all `PREVIEW` queries are trivial to recognize since both touch points are on the same object. In terms of accuracy, the baseline gestural recognition fails to recognize 2 of the 15 queries tested. Given schema information, our classifier correctly recognizes 100% of the queries (and implicitly, query types). As shown in Figure 4, using the data distribution information improves *prediction* ability for

`JOIN` queries, and has no impact on `UNION` queries, since the `UNION` operator does not leverage the data information for disambiguation.

Related work

Gestural interaction in domain-specific use cases has been studied [13] widely. User-friendly solutions to interacting with databases have ranged from example-driven querying [1], to automated form generation [7] spreadsheet interfaces [2] to autocompletion [8, 9] and query recommendation [3]. Visual analytics systems such as Tableau [12], TaP [5] and SQL Server Kinection [10] map interactions and gestures from the UI layer to a set of database query templates, without considering the contents of the database itself. Probabilistic methods to improving gesture recognition[14] and mapping interaction to actions [4, 11] have been discussed before, however such methods would be too computationally intensive to recognize the space of all possible database queries. In contrast to these systems, our classifier performs a two-stage recognition, mapping gesture coordinates to action types, and then further using an arc layout and database statistics to successfully identify the exact query.

Ongoing Work

We plan to evaluate the usability and learnability of our system over users at multiple levels of proficiency of querying databases and our gestural interface. Gesture coordinates from these user studies will then be used to further tune the parameters of our classifier, which can then be evaluated using k-fold cross validation to measure the benefits (in accuracy and prediction) and generality (across both users and queries) of user training. Finally, we plan to evaluate the impact of providing insights to the user in the form of result previews for the most likely query *while* the user is articulating the gesture.

References

[1] Abouzied, A., et al. Dataplay: interactive tweaking and example-driven correction of graphical database queries. *UIST* (2012).

[2] Bakke, E., Karger, D., and Miller, R. A spreadsheet-based user interface for managing plural relationships in structured data. *CHI* (2011).

[3] Chatzopoulou, G., et al. Query recommendations for interactive database exploration. *SSDBM* (2009).

[4] Damaraju, S., and Kerne, A. Multitouch gesture learning and recognition system. *Tabletops and Interactive Surfaces* (2008).

[5] Flöring, S., and Hesselmann, T. Tap: Towards visual analytics on interactive surfaces. *CoVIS* (2010).

[6] Jagadish, H., et al. Making database systems usable. *SIGMOD* (2007).

[7] Jayapandian, M., et al. Automating the design and construction of query forms. *TKDE* (2009).

[8] Khoussainova, N., et al. Snipsuggest: context-aware autocompletion for sql. *VLDB* (2010).

[9] Nandi, A., and Jagadish, H. Assisted querying using instant-response interfaces. *SIGMOD* (2007).

[10] Patney, S., et al. Sql server kinection. *PASS* (2011).

[11] Schwarz, J., Mankoff, J., and Hudson, S. Monte carlo methods for managing interactive state, action and feedback under uncertainty. *UIST* (2011).

[12] Stolte, C. Visual interfaces to data. *SIGMOD* (2010).

[13] Ullmer, B., and Ishii, H. Emerging frameworks for tangible user interfaces. *IBM Sys. Journal* (2000).

[14] Weir, D., Rogers, S., Murray-Smith, R., and Lochtefeld, M. A user-specific machine learning approach for improving touch accuracy on mobile devices. *UIST* (2012).