

GestureQuery: A Multitouch Database Query Interface

Lilong Jiang Michael Mandel Arnab Nandi
Computer Science & Engineering
The Ohio State University
{ljianglil,mandelm,arnab}@cse.osu.edu

ABSTRACT

Multitouch interfaces allow users to directly and interactively manipulate data. We propose bringing such interactive manipulation to the task of querying SQL databases. This paper describes an initial implementation of such an interface for multitouch tablet devices called *GestureQuery* that translates multitouch gestures into database queries. It provides database users with immediate constructive feedback on their queries, allowing rapid iteration and refinement of those queries. Based on preliminary user studies, *GestureQuery* is easier to use, and lets users construct target queries quicker than console-based SQL and visual query builders while maintaining interactive performance.

Categories and Subject Descriptors

H.5.2 [Information Interfaces and Presentation]: User Interfaces

1. INTRODUCTION

Console-based SQL interfaces are difficult to use, especially for non-experts, especially in situations where the database query language, the underlying data or the schema are unfamiliar to the user. While there has been a significant amount of work towards making databases usable to end users [11], ranging from example-driven querying [7, 10, 17] to user interface innovations such as autocompletion [14], they are typically geared towards keyboard-driven interfaces. Some mature products, such as visual query builders, are already part of several database management systems. For example, Microsoft Access and *pgAdmin* [4] allow users to build SQL in a visual manner by interacting with a specialized UI. These products are designed for keyboard-and-mouse settings and do not provide immediate feedback to guide users towards the intended query. With the popularity of mobile and interactive-computing devices [8], such as iPad, Kinect, and Google Glass, there has been a rise in the access, manipulation and querying of data on such devices. To this end, we have previously proposed a new database architecture

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... \$ 10.00.

system geared towards keyboard-less database interaction, called *GestureDB* [13]. There are many challenges in gestural data access, from creating an intuitive and complete query language to accurately processing query intent and generating relevant feedback for the user. *GestureQuery* is one possible frontend to *GestureDB*, making it possible for users to construct and execute queries over relational data using a multitouch interface. As described in the user studies, *GestureQuery* presents better usability compared to the traditional query interfaces, both console-based and visual.

2. A MULTITOUCH QUERY INTERFACE

2.1 Interface Layout

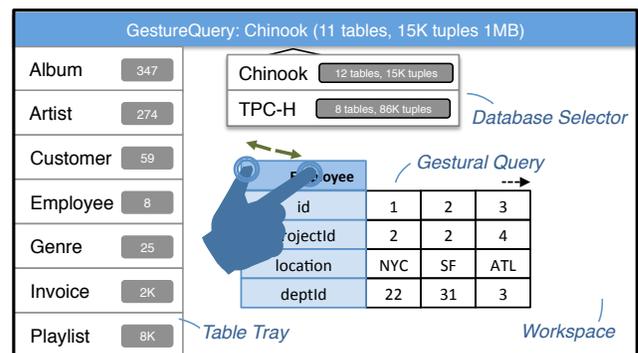


Figure 1: The *GestureQuery* user interface. Users can select a database, drag tables from the tray on to the workspace, where they can perform a series of gestural queries directly on the table data.

The multitouch database interface, as shown in Figure 1, is divided into three parts: the header, tray and workspace. The header displays the database information and allows users to pick another database. The tray shows a list of tables available in the selected database, along with tuple cardinalities. It also acts as a source for database table interface objects. To begin building a query, the user drags a table from the tray into the workspace. Note that this is a “clone” operation – multiple copies of a table can be involved in a query by dragging copies repeatedly to the workspace. A table is displayed in the workspace as a stack of rectangles, the first rectangle showing the table name and subsequent rectangles showing its attributes. Users perform query operations by manipulating the tables in the workspace using a series of multitouch gestures.

Employee			
id	1	2	3
projectId	2	2	4
location	NYC	SF	ATL
deptId	22	31	3

Figure 2: The PREVIEW gesture

Employee			
id	1	2	3
projectId	2	2	4
location	NYC	SF	ATL
deptId	31	3	

Figure 3: The FILTER gesture

Employee			
id	1	2	3
projectId	2	2	4
location	NYC	SF	ATL
deptId	22	31	3

Figure 4: The SORT gesture

2.2 Interactive Direct Manipulation

Our system allows users to directly manipulate queries[16], two system properties are required to do so. First, users must be able to manipulate the data with their gestures directly in a visual way. Secondly, the system must provide instantaneous feedback, which can assist users in either constructing their intended query or exploring the space of possible queries in a fluid manner, not interrupting the user’s flow of thought.

2.3 A Gestural Query Language

Interaction and Language Design: We now describe the gestural query language used by GestureQuery. The query language works on the relational model and follows a *gestural query paradigm* [13], in which the user has a vague query intent, and articulates the intent by performing a gesture. Gestures are designed to be **multitouch**, allowing operations to scale to variable amounts of complexity, by using additional touch points. As the gesture progresses, the intent transitions from vague to fully articulated. To aid the user in articulating the intent, the interface attempts to **infer possible queries** during the gesture, and provides **constant feedback** to aid the user. Upon recognizing an unambiguous query intent, the UI transforms the workspace to present the new table. Based on this interaction pattern, we use a query algebra that is **closed**, such that each **operation** in our algebra takes one or more relations, and produces another relation. This allows the user to stack multiple operations, allowing for complex queries to be performed. Based on this interaction paradigm, we now present a set of query operations, each of which performs a transformation on one or more relations. Each operation consists of a multitouch **gesture** denoting the operation, instantaneous **feedback** given to user, and the final **result** of the transformation.

PREVIEW: As shown in Figure 2, this operation is implemented by a pinch-out gesture on the table header. The result of this operation is to present the user a certain number of tuples. When the user puts two fingers on the table header, the system will recognize this may be a potential pinch-out gesture so the initial distance between these two fingers is recorded. Every time a *touchMove*¹ event is triggered and the current distance is greater than the initial distance by a threshold, the system recognizes a pinch-out gesture and shows the table’s first set of tuples. After the table is previewed, the user can browse the data by scrolling horizontally through the table content, as shown in Figure 2. The GestureDB database layer utilizes prefetching and lazyloading strategies to cache results at the frontend.

¹touchMove, longTap, swipeRight and swipeLeft multitouch events are defined in the iOS & ZeptoJS mobile touch APIs.

FILTER: As shown in Figure 3, this operation allows the user to select all tuples with a matching value of a particular attribute. When the user *longTaps* a table cell (or multiple cells, in the case of multitouch), the system will recognize that this may be a filter operation. A free-floating copy of the cell appears under the user’s finger as feedback, and if the user drags that cell to the corresponding field header, the tuples will be filtered to match the value(s) selected.

SORT: Figure 4 describes the SORT operation that allows the user to sort the tuples in a table by the values in one attribute in either ascending or descending order, implemented using *swipeRight* and *swipeLeft* touch events. When the user swipes an attribute header from left to right, the table will be sorted in an ascending order by that field. When the user swipes an attribute header from right to left, the table will be sorted in descending order. Due to the ephemeral nature of this operation, the only feedback provided is at the UI level, that of the attribute labels moving.

REARRANGE: Faithful to the original definition by Codd [9], the gesture shown in Figure 5, allows the user to rearrange the attributes in a table by dragging the attribute header into the intended position.

GROUP BY AND AGGREGATE: This pair of operations allows the user to group the tuples by values of a particular attribute, and aggregate using a user-selected function. Both gestures are shown in Figure 6: in order to perform a **GROUP BY AND AGGREGATE**, the user drags two attribute headers at the same time. Once the grouping attribute header overlaps the table title, an aggregation menu consisting of the potential aggregation functions will show up near the table. The user then drags another attribute header onto one aggregation functions in the aggregation menu to aggregate on that attribute. Should the grouping operation yield a large number of groups (i.e. greater than a certain threshold, set to 1K for our UI), the UI warns the user about the aggregation by flashing the table in red as feedback.

JOIN: An equijoin operation is performed by dragging two tables towards one another. The system attempts to determine which attributes to join the tables on based on their proximity and compatibility with one another [15]. To aid in this process, the system arranges the attributes of each table in an arc that the user can adjust. Specifically, the user can perform a four-finger gesture, with two fingers on each table, to move the center of the arcs and adjust the attributes that are closest to one another. Once the system recognizes the JOIN operation, feedback is provided in the form of a join preview to give user insight into the data, as

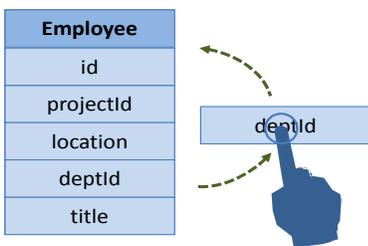


Figure 5: REARRANGE

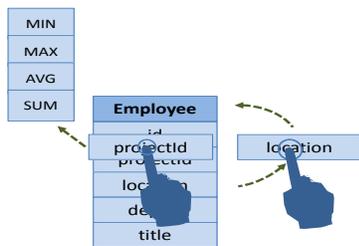


Figure 6: GROUP BY / AGGREGATE

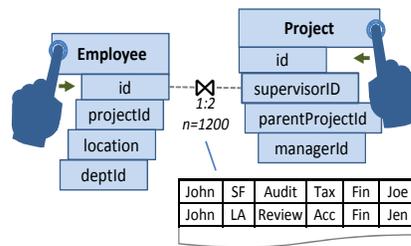


Figure 7: JOIN

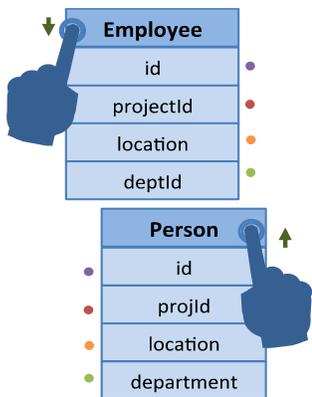


Figure 8: The UNION gesture

shown in Figure 7. If the user continues to move the tables closer, the preview table will become a regular result table, allowing for further gestural queries on the result.

UNION: To perform a UNION operation, the user moves one table on top of the other in a stacking gesture, as shown in Figure 8. We determine whether it is a stacking gesture through calculating the distance between the last attribute header in the upper table and the table header in the bottom table. For feedback, the colors beside the tables indicate the attribute pairs that are schema-compatible.

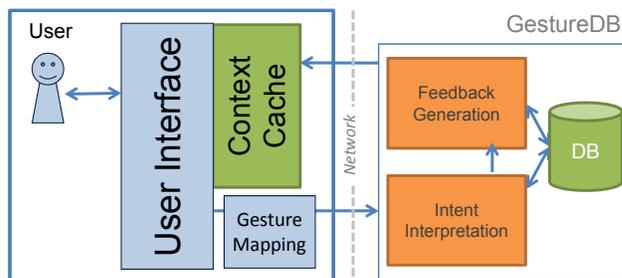
Resolving ambiguity: Since a two-table gesture may be a JOIN or UNION operation, we use a maximum entropy classifier to distinguish these two operations. This classifier uses two features, proximity and schema compatibility to predict the likelihood of all possible JOIN and UNION queries between the two tables. The proximity feature measures how close these two tables or attributes are, while the compatibility feature describes whether two tables are schema compatible for UNION or two attributes are of the same data type for JOIN. In order to avoid unnecessary computations, the classifier is only run when the distance between the two tables is below a threshold, and the computation of the likelihood of each potential query is only carried out until an answer can be found. For example, if two fields are not schema-compatible, their distance will not be computed.

3. ARCHITECTURE

We implemented the GestureQuery interface on iPad in javascript using the ZeptoJS library [6], connecting to the backend using Tornado [5]. When DOM elements register for multitouch events, DOM elements will continually

receive TouchEvent objects (The TouchEvent class encapsulates information about a touch event [2]) as fingers touch and move across the iPad. Our system classifies the gesture into the intended query according to the multitouch coordinates collected so far. In order to avoid extra computation, the classifier performs distance calculations and looks into schema features only if there is the possibility of an ambiguous gesture. GestureQuery additionally minimizes network communication with the backend database by utilizing a client-side cache for both schema and data.

GestureQuery



4. USER STUDY

In order to evaluate the usability of our system, we ran a preliminary user study, comparing GestureQuery against console-based SQL and a visual query builder. In order to evaluate all components of *GestureQuery* (i.e. the UI, the query language, the maximum entropy classifier and the feedback generated from the backend), we consider the two complex operations: JOIN and UNION – all other operations are either equivalent or drastically simpler in our UI. In this experiment, we use Active Query Builder [1] as the visual query builder, since it allows the construction of both JOIN and UNION queries.

Five users, at varying levels of database expertise (3 CS grad students, 2 non-CS grad students) were instructed to perform 4 query tasks for all three systems, with the order of tasks and systems randomized. Two query tasks involved JOIN queries and the other 2 were for the UNION query. The target query was provided explicitly posed to them as an English query, including the exact names of the schema elements, such as “Perform a JOIN between Album and Artist ON ArtistID”. Users were provided a brief overview of all systems prior to usage, and were asked after the tasks if the GestureQuery frontend was (1) easy to learn (2) easy to use after the tasks.

Metrics: We recorded the time that it took each user to perform each query with each system from the time that they

started the interface to the time that the system correctly recognized their intended query, referred to as *prediction time*. For our system, we also recorded the time that it took our system to make a decision after *each* TouchMove. This measure, named **performance**, is used to determine whether the system can be used interactively or not. It should be noted that the *GestureQuery* interface poses the distinct disadvantage over the competing interfaces in that it lacks a keyboard or mouse, which are typically considered required for querying.

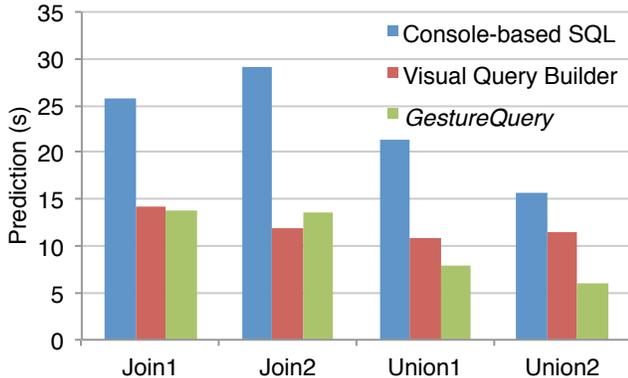


Figure 9: Time-to-task (prediction time) for non-trivial queries, for each system, averaged across 5 users. The multitouch GestureQuery interface is better / comparable to competitors.

Results: Qualitatively, users agreed that GestureQuery was either faster or comparable with the visual query builder to learn, and mostly better than the visual query builder to use. Figure 9 shows the average prediction time for each task. GestureQuery allows users to create their desired queries much more quickly than console-based SQL in all experiments, and is either faster than or comparable with the visual query builder.

Join1	Join2	Union1	Union2
1.042%	1.175%	2.339%	3.211%

Table 1: Performance Times over 33ms

Based on an ideal threshold of 30 frames (i.e. touch events) per second to ensure fluid interaction, we enforce a goal of 33ms on the per-touch execution time of our system. Table 1 shows the percentage of touch events where our UI responded in time greater than 33ms. As we can see, the percentage is low enough to allow interactive performance. In fact, almost all performance times except 2 touch events for each gesture were less than the threshold; these corresponded to the time when the system is waiting on the backend server to respond.

5. DEMONSTRATION

For our demonstration, we will present to users an app on the Apple iPad. The app communicates over WiFi with a backend, hosted on a single PC. Users can perform all described operations, including PREVIEW, FILTER, SORT, REARRANGE, GROUP BY & AGGREGATE, JOIN and UNION. Users can get instantaneous insights into the data through the feedback provided by the system for each operation, and thus can be assisted in creating their intended query. Users

will be able to pick from multiple publicly available relational datasets, such as Chinook [3], an open-source digital media store database and one based on TPC-H to represent ad-hoc, decision support datasets.

6. CONCLUSION AND FUTURE WORK

This paper describes an initial implementation of the GestureQuery iPad front end for GestureDB. It allows users to formulate queries more quickly than existing interfaces while providing fluid, real-time feedback to facilitate schema and data exploration. We have shown that it is possible to implement a complete SQL algebra through gesture and that it can be applied to querying databases without keyboards from the gesture-based interfaces of the future.

In the future, a comprehensive user study is needed to be done over all the actions. We will add additional operations to enrich the query space, such as projection and the range filter. We will add convenient functions such as undoing recent operations and deleting tables. And finally, we will simplify the JOIN further so that even in the hardest cases it is faster than existing systems.

7. ACKNOWLEDGEMENTS

We would like to thank NEC Laboratories America for supporting part of this work.

8. REFERENCES

- [1] Active Query Builder. <http://www.activequerybuilder.com/>.
- [2] Apple Developer. <http://developer.apple.com/>.
- [3] Chinook Database. <http://chinookdatabase.codeplex.com/>.
- [4] pgAdmin. <http://www.pgadmin.org/>.
- [5] Tornado. <http://www.tornadoweb.org/>.
- [6] Zepto. <http://zeptojs.com/>.
- [7] A. Abouzieed et al. Dataplay: interactive tweaking and example-driven correction of graphical database queries. *UIST*, 2012.
- [8] Canalys. Worldwide Smartphone and Client PC Shipment Estimates. 2012.
- [9] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 1970.
- [10] Y. Ishikawa et al. Mindreader: Querying databases through multiple examples. *VLDB*, 1998.
- [11] H. Jagadish et al. Making database systems usable. *SIGMOD*, 2007.
- [12] D. Kammer, M. Keck, G. Freitag, and M. Wacker. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. *EPMI*, 2010.
- [13] A. Nandi. Querying Without Keyboards. *CIDR*, 2013.
- [14] A. Nandi and H. V. Jagadish. Assisted Querying using Instant-Response Interfaces. *SIGMOD*, 2007.
- [15] A. Nandi and M. Mandel. The Interactive Join: Recognizing Gestures for Database Queries. *CHI*, 2013.
- [16] B. Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. *IUI*, 1997.
- [17] M. M. Zloof. Query by example. *NCCE*, 1975.