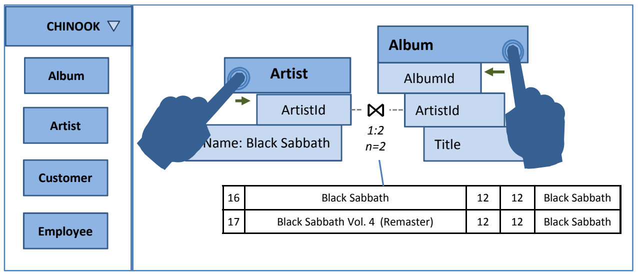
Gestural Query Specification

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

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 modes of interaction, database query interfaces require a fundamental rethinking to work without keyboards. We present a novel query specification system that allows the user to query databases using a series of gestures. 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. We conduct exhaustive systems performance tests and user studies to demonstrate that our system is not only performant and capable of interactive latencies, but it is also more usable, faster to use and more intuitive than existing systems.



16	Black Sabbath	12	12	Black Sabbath
17	Black Sabbath Vol. 4 (Remaster)	12	12	Black Sabbath

Figure 1: Query Specification using Multi-touch Gestures. The query task from our motivating example (Section 1.1) is specified using a series of 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 preview of most likely join is presented as feedback to the user for guidance.

1. INTRODUCTION

Next-generation computing devices such as tablets, smartphones, motion capture-based systems such as the Kinect, and eye-tracking-based systems such as Google Glass have ushered us into a new age of *natural user interaction* with data. The number of smartphones and tablets sold in 2011 was 1.5 times the number of desktops, laptops, and netbooks combined [9] for the same period; and in the last quarter of that year, this ratio jumped to 1.9 times. In a more recent study [29], sales of tablet devices are expected to surpass those of portable computers in 2013, and surpass *all* personal computers in 2015. Based on these trends, it is clear that both the size and the heterogeneity of non-keyboard interaction is growing rapidly, and soon will be a dominant mode of interaction.

End-user-friendly interaction with databases is a well-motivated problem [31]. As discussed in the related work section (Section 6), there has been a wide variety of work in query interfaces in both domain-specific and domain-independent use cases. However, current efforts are based on keyboard or mouse-driven interaction, and are therefore unsuitable for gestures.

We capture the heterogeneous modes of natural, movement-based interaction with data under a common terminology, **gestural querying**. Each *gesture* is a single movement – performed by the user using a finger, hand, head or input device, denoting an action to be performed on the database query interface. Gestures encompass finger-based multitouch points (e.g., input from the iPad) including pressure for each finger, skeletal motion tracking (e.g., Kinect), and more. We label databases that support such querying *gestural databases*. In this paper, we articulate the challenges in building a query interface that supports gestural querying from such inputs, and study a novel system that allows for gestural query specification.

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, that identifies the gesture as one of a set of gestures predefined by the operating system, independent of the application state. The gesture type, along with metadata parameters such as *coordinates* and *pressure*, are sent to the application, which then uses them to infer actions. This mapping of interpreted gestures to a domain-specific action can be considered as a *classification problem.*

Unlike domain-specific applications, 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

Proceedings of the VLDB Endowment, Vol. 7, No. 4
Copyright 2013 VLDB Endowment 2150-8097/13/12.

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

of query, and then drill down the parameters of the desired query. Such a modal interface goes against the desiderata of fluidity and direct manipulability. Further, if the user is unsure about the exact query [44] itself, the database system will need to guide the user to the exact query intent, and the intended query result.

Thus, to provide for an effective querying experience, there is a strong need for the system to effectively aid the user in articulating the intended query using gestures. This problem of **gestural query specification** relies on both gestural input and the state of the database to recognize and articulate queries.

1.1 Motivating Example

To better explain the system, we motivate our problem with a simple ad-hoc querying task. Consider a relational database about music, which contains information about music artists, albums and performances. From this database, the user is tasked to ***find the titles of all the albums created by the artist "Black Sabbath"***. While the task is simple to express, it embodies many of the challenges users have with querying the system – discovering the schema and data, understanding the query language and interface, and then interacting with the system to extract the required information.

Figure 1 demonstrates this task using one possible frontend for gestural querying – a multitouch interface. The user first browses the schema and data by dragging elements from the database tray into the workspace, and then peeks into them by pinching out on each table. The artist *"Black Sabbath"* is filtered by using a simple gesture: dragging the "Black Sabbath" cell to the attribute header, followed by a join operation, performed by bringing the ARTIST and ALBUM close to each other. The interface fluidly guides the user by responding to each touch and gestural movement, providing previews and feedback of the results to the most likely query. It should be noted that the interaction is done directly on the data itself, and is composable – queries can be constructed piecemeal, and the ordering of the filter clause and the join could be interchanged. The gestural query paradigm allows for the user to intuitively specify an articulate query by interacting with the system without having to invest time learning about the schema, data, or the query language. As we will see in our experimental evaluation, our novel query specification system outperforms existing methods in usability.

1.2 Challenges and Opportunities

The use of gestures to support ad-hoc database interaction brings forth a set of challenges and opportunities. **Interaction is continuous** – unlike traditional keyboard interaction where the user types out a string of characters (denoting a query) and then pressing enter to execute it, gestural interaction is expected to be fluid. Users expect to see feedback *during* the articulation of the gesture itself. This leads us to observe the emergence of a new **database query paradigm**: in place of the traditional *Query→Result* interaction, we now have a continuous loop of interaction, where the user receives constant feedback from the database during the specification of the query itself. Second, touch-based, augmented reality, and motion-oriented interfaces have popularized the notion of a **direct manipulation** [27] interface. In the context of database querying, users expect to manage, query, and manipulate data by directly interacting with it – a contrast from the current paradigm of constructing declarative queries and then applying them to the data. Finally, new patterns of interaction are likely to overwhelm the user – unlike traditional keyboard interaction with a limited number of keys to press and keywords to generate, there can be an infinite number of gestures, and implied actions from these gestures. Thus, it becomes the responsibility of the query specification system to **guide the user** during the database interaction process.

Contributions & Outline: In this paper, we make the following contributions:

- We introduce and define the problem of *gestural query specification*, a critical task in database querying that is gaining importance with the proliferation of purely gesture-oriented interfaces.
- We propose a novel querying framework that allows the user to effectively specify relational queries using a series of gestures.
- We detail the architecture and implementation of a novel gestural query specification system, GESTUREQUERY, that provides feedback to the user during the query process, and detail the various systems considerations and optimizations involved in the building of the system.
- We evaluate the effectiveness of our system by performing exhaustive usability studies and systems evaluation using real workloads. We show that GESTUREQUERY is not only performant to use for real-world use cases, but also significantly more usable compared to existing querying methods.

The paper is organized as follows. In the following section, we describe the various terminologies and preliminary concepts that are central to our work. In Section 3, we describe a novel *Gestural Query Language*, that allows the user to specify relational queries using an intuitive set of gestures. Section 4 details the underlying architecture and implementation considerations of our *Query Specification System.* In Section 5 we provide a detailed performance analysis and discuss the results of an exhaustive usability study. We end with related work, conclusions, and future work.

2. MODELS AND PRELIMINARIES

Data Model: It should be noted that while our paper uses the *relational model* as its underlying data model, our contributions are independent of this choice, and can be applied to any other data model. From a presentation standpoint, we additionally allow relations to have `ORDER BY` attributes.

Query Model: Our system allows for both read and write queries, at the data and the schema level. Specifically, we allow for selection, projection, union, join, and groupby/aggregate operations over relations. Selections are constructed as conjunctions of clauses, and disjunctions can be created using the union operator. As we will describe in Section 3, each query operation takes one or more relation as input, and outputs one relation. In addition, each operation generates *feedback* for the user during the gesture articulation phase to guide them.

Gestural Articulation: A *gesture* is defined as a movement of the hand or body. In our system, a gesture G is represented as a set of time-series of points; $p_i = \langle t, l, m \rangle$ where t is the timestamp of that point, l is locational information in the interface, and m is the metadata information associated with that point and time and $|p|$ is the number of points. For example, in the context of a capacitive 2-dimensional multi-touch interface that supports 10 fingers at a touch sampling rate of 30Hz (such as those found in most tablet devices), there would be a time-series of up to ten 2-dimensional points, at 30 sets per second: $l = \{\langle x, y \rangle\}$ and $|l| \leq 10$. m in this example could represent *pressure* and *size* of the finger touch. Our model of gestural input allows for a wide variety of input: for 3-D depth-capture devices such as the Microsoft Kinect, $l = \{\langle x, y, z \rangle\}$. The user interacts with the system in a series of **articulations**, individual fluid motions. Each gesture articulation typically lasts a few seconds, going from an ambiguous gesture at the start (the user has not provided any point information yet) to an articulate gesture (the user has made a discernible gesture, encoded as hundreds of timestamped point instances), followed by the completion of the gesture. It should be noted that for gestural interfaces, the user expects feedback from the system *during* the articulation of the gesture as well – it is not sufficient to only react at the end of the articulation. This is because the nature of feedback might affect the gesture the user performs. For example, a user who is trying to perform a `UNION` on two relations may notice that the schema of the two relations are in fact incompatible, and thus consider performing a `JOIN` instead, changing from a stacking gesture (one relation on top of another) to a composing gesture (two relations side by side) in the same fluid finger movement.

Query Specification: Given a database query task, the user performs a series of interactions to query the database. The process of interacting with the database to go from a conceptual task to specifying the correct structured query is called query specification. Query specification solves a query task, and comprises one or more *query intent transitions*.

Query Intent: Users issue queries by focusing on their query intent, going from a vague information need to an exact query formulation. The query intent is a probability distribution over a space of valid relational queries. Initially, the user provides no information to the system, thus the query intent is a uniform distribution over all possible queries to the database. Given sufficient information from the gesture and the query context (described below), the system can narrow down this distribution, bounding the space of possible queries. At the completion of a gesture, the query intent space is that of a single query with 100% probability: an explicit query. The process of narrowing the query space and arriving at an explicit query is called a *query intent transition*. A single user session will comprise multiple query intent transitions, each building on the previous one.

Query Context: All interactions take place within a query specification session. Due to the directly manipulable nature of the interface, the system catalogs all query intent transitions and all recent intermediate results and feedback, termed the *query context*. This allows the system to infer and narrow the space of possible queries. Further, the query context can be used to prioritize the surfacing of feedback to the user, as discussed in the forthcoming paragraphs.

Query Gesture: Each set of user gestures is codified as a search pattern, with a "likelihood score". Each gesture maps to one or many parameterized queries. When the likelihood score for a certain gesture goes above a fixed threshold, the system attempts to populate the parameters of the parameterized query using the gesture information (e.g., which relations are being dragged) and the query context, building a query template. This query template is used to narrow the query intent space (described above). In the event that multiple gestures are inferred at the same time, the intent space is the union of all corresponding query templates.

Intent Feedback: The goal of the user interface is to accelerate the narrowing of the intent space, allowing the user to quickly reach an explicit query. To do this, the system provides feedback to the user during the entire interaction loop. Since the amount of feedback possible is quite large, and there is a cost of overburdening the user with too much information. This leads to an interesting problem: How do we rank feedback such that it causes the user to narrow the intent space? While the presentation of feedback is central to the user interaction, the generation of feedback itself can leverage prior work in areas such as result diversification [55] and is beyond the scope of this paper. In this paper, we focus on the query language, interaction paradigm, and query specification system.

Example Walkthrough: To familiarize the reader with the terms presented, we walk through the user interaction in the motivating example. The user has the **query specification** task of representing a `FILTER` (i.e., selection) followed by a `JOIN`. This is done using a series of six quick **gestures** – the user first performs a gesture to drag the ARTIST relation into the workspace, putting it in the **query context**, followed by a pinch-out gesture to `PREVIEW` the relation. The same pair of gestures is performed on the ALBUM relation. The user then performs a `FILTER` gesture on ARTIST, selecting only "Black Sabbath". Due to the closed algebra, all direct manipulations on presented data result in relations as well, thus allowing subsequent actions on the data. The user then articulates a `JOIN` gesture by bringing the relations close to each other, considering each pair of attributes. *During* the gesture articulation, the **query intent** is initially equally spread across all the valid `JOIN`s – for tables with M and N attributes, this can yield $M \times N$ possible queries. The user is provided with **intent feedback**, a preview of the most likely join result and join statistics. The user then uses this feedback to complete the articulation of the query, bringing the most desirable pair of attributes close to each other. This articulation completes the **query intent transition** – the intent has gone from a uniform distribution over $M \times N$ possible queries to exactly one possible join query. This completes the articulation of the gesture, and correspondingly, the intent transition and the overall specification of the tasked query. Ultimately, the user is left with the desired result.

3. A GESTURAL QUERY LANGUAGE

As depicted in Figure 1, the database query interface allows users to directly manipulate results by interacting with them in a sequence of gestures. The interface 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. The tray also acts as a source for database table interface objects. 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 relational queries, all actions manipulate one or more relations into another relation. Thus, actions are composable and can be performed in sequence, manipulating relations in the workspace till the desired result is achieved.

3.1 Desiderata

The development of a gestural query language is a critical task – the language needs to capture the nuances of the new generation of user interfaces, while at the same time, provide the user with the expressiveness and capabilities of existing querying methods. As a process, we detail the desired features of an ideal gestural query language. As we will see in the following sections, we leverage a combination of **algebraic**, **algorithmic**, and **design-based** solutions to meet these needs, and come up with a usable and efficient way to specify queries using gestures.

Direct Manipulation: Modern user interfaces have brought forth a paradigm shift in user input and output. In place of indirect interaction via textual commands or mouse movements, users now expect to touch, move, and interact *directly* with the data and interface elements. Therefore, our system needs to allow for direct manipulation:

Users should not be expected to construct abstract queries, but to interact with the data directly. Thus, queries are constructed *implicitly* – each gestural operation by the user transforms the existing data, iteratively specifying the intended query in steps.

Closure and Composition: The iterative and directly manipulable nature of the system implies that the data presented to the user can be perceived as a view of a query expression over the database. For interaction to be intuitive, all gestures (and thus the corresponding query operations) should be available to the user at all times. Therefore, we require that our gestural query operators be **closed** under relational algebra, and thus all gesture operations are performed on one or more relations, and yield new relation as the final output, allowing for composition of operations.

Feedback: While each gesture is performed on a relation and yields a relation upon completion of the gesture, it is imperative that the user be *guided* through the space of possible queries and results. As mentioned in the previous section, this is increasingly important in the context of gestural interfaces where the space of possible gestures is unbounded and there are few constraints provided to the user (e.g., finite number of keys on a keyboard versus a touch interface with no explicit targets). Unlike the final result of each operation, feedback need not always be a relation – it can be any piece of information that aids the user in articulating their intended query. From a usability perspective, we observe that modeling feedback on the most likely query result is most useful to the user.

Expressivity: It is important that our language should not hold the user back in terms of their ability to specify queries. In the following subsection, we present our gesture vocabulary that allows users to perform both schema and data-level queries, providing a mapping between gestures and most relational algebra operations. Further, it is important for the system to let the user express queries efficiently – thus, most gestures allow taking advantage of ideas such as multi-touch to perform complex operations (e.g., conjunctive selections can be performed by filtering on multiple values and attributes, one per finger, that can then be `UNIONed` together to return a disjunction of conjunctive selections).

3.2 A Gesture Vocabulary

In this section, we describe the vocabulary of gestures used in our system. Our vocabulary works on the relational data model and assumes the use of multi-point gestures, as per our gesture definition. It should be noted that while the existence of a gesture vocabulary is critical to our system, the *gesture itself is not important* – users are welcome to come up with their own creative and intuitive gestures. Custom gestures, when encoded as feature functions for the gesture classifier will work similar to the gestures proposed below. However, as discussed in Section 3.3, our gesture vocabulary has been carefully thought through and has been through several iterations of design. As part of the overall system, this set of gestures has been formally evaluated to be highly usable and intuitive as per the evaluations performed in Section 5. Visual representations of most actions are presented in Figure 2.

`UNDO:` This is a universal action, performed by swiping the *workspace* to the left. This restores the workspace to the previous state, undoing the prior gesture. This (non-operator) query is implemented by maintaining a stack of prior queries; and is extremely useful for exploration / trial-and-error query specification tasks.

`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 a SQL query `SELECT * FROM TARGET_TABLE LIMIT 6;`, presenting the first six rows of the table on screen[2].

`SORT:` By swiping the attribute header right/left, users can sort the data according to the attribute in ascending/descending order. The sorted attribute is highlighted. Ordering by multiple attributes can be performed by successively swiping headers.

`AGGREGATE:` This action works on a single table. The user first needs to drag the grouping attribute to the table header to bring up a popup menu of aggregate functions. The user then drags the aggregated attribute through the desired aggregation functions in the menu.

`REARRANGE:` By dragging the attribute header into a different place in the attribute list, the user can change the relative positions of the attributes. This allows positional operations such as `JOINS` to be performed more easily.

`FILTER:` This action works on a single table. By tapping and holding a table cell, a free-floating copy of the cell shows up under the user's finger. The user can drag this copy into the attribute header to filter the preview. If the user release his finger immediately after the copy overlaps with the attribute header, the table will be filtered by equality to the selected value. If the user holds the copy on the attribute header, a range slider will appear and the user can filter the table by adjusting the range on the slider.

`JOIN:` Two tables can be joined 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. When two tables are brought close to each other, their attribute lists curve so that the user can bring the desired pair of attributes closest to each other. The design considerations for this curvature are described in Section 3.3.

`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 compatible columns is presented as color coded feedback.

`INSERT:` In order to insert one new tuple, the user can doubletap the bottom of the table. A blank tuple will be added into the end of the table that the user can edit.

`UPDATE:` To update the value of a table cell, the user uses two-fingers to swipe the cell. If the cell is numeric, scrolling up will increase the value while scrolling down will decrease the value. If the user holds their finger on the cell, a distribution slider will pop up and the user can update the value through the slider. If the value is categorical, a menu will appear allowing the selection of a category. If the value is alphanumeric, text input with autocompletion will be triggered, based on prior gestural text input work [56].

`ALTER:` There are two ways of changing the table schema. One way is to change the attribute layout using the `REARRANGE` action. The other way is to scroll the attribute header using two fingers, which allows the user to change the datatype or name of the attribute.

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

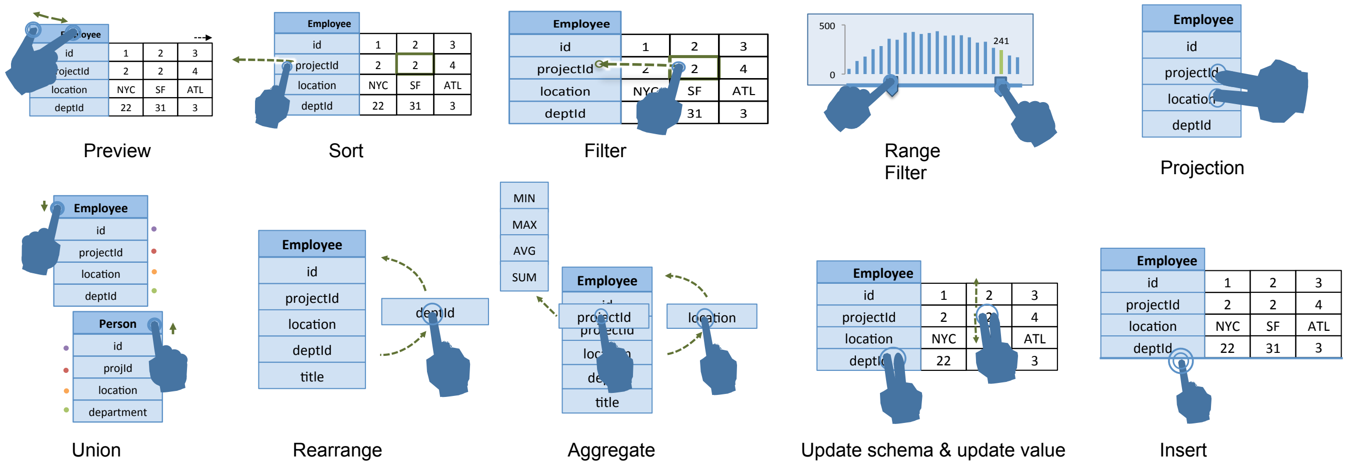


Figure 2: Gestures in the GESTUREQUERY gestural query language. Usability experiments demonstrate that this language is both easy to use and quicker to use than traditional interfaces.

`PROJECTION`: The user can drag multiple attributes out of one table to make a new table.

3.3 Design Considerations: Interactive Join

It is important to note that while the above vocabulary of gestures is demonstrative and gestures can be trivially replaced with others, the design of these gestures was done with care and is the product of multiple iterations of user testing. To provide insight into the development process of these gestures, we now walk through two design aspects of the `JOIN` gesture.

3.3.1 *Minimizing Gestures for Exploratory Querying*

An important challenge in ad-hoc querying is that the user is typically unfamiliar with both the schema and the data. Thus, we expect the user to perform a large number of trial and error queries, discovering the database in the process. To encourage the user to discover the database, and at the same time minimize the effort involved, we consider a variety of gestures for `JOIN` (shown in Figure 3) and pick the one that is intuitive while quantifiably involving the least amount of effort for exploratory querying.

One possible gesture (Figure 3 (a)) for `JOIN` is inspired by mouse-driven interfaces for SQL databases – the user can simply drag a column from one relation to another. The system can generate a preview of the join during the drag operation, allowing the user to abort the join if the result does not look right. However, for relations with M and N attributes each, finding the *right* combination of attributes will involve a worst case of $M \times N$ *separate* drag gestures, each considering only one join preview per gesture.

A variant of this gesture (Figure 3 (b)) is to allow the user to not abort the gesture on an unfavorable preview result, and to continue the drag gesture towards a different column. This allows the user to discover the right join in $\text{MIN}(M, N)$ gestures, which is a reduction from the previous number of actions.

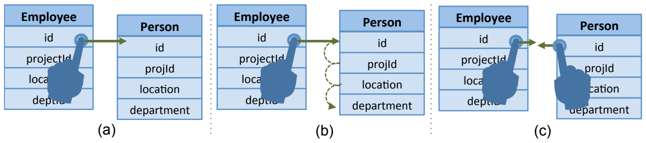


Figure 3: Considerations for the interactive `JOIN`. Each of these options requires more gestures than the one implemented in our system, as seen in Figure 1.

A third method (Figure 3 (c)) is to pick two attributes in each of the relations and then to drag them together in a pinching action. Clearly, since we begin the gesture by picking the two attributes, we are again forced to perform a worst case of $M \times N$ gestures, which is again undesirable.

The fourth method is to simply bring the two tables close to each other. By moving each attribute close to the other *during* the gesture articulation phase, there is exactly *one* dragging gesture to perform, and all possible attribute combinations can be previewed in the same gesture. Thus, this gesture is the one we use for `JOIN` queries, shown in Figure 1. As we will see in our experimental evaluation, this JOIN gesture is both quicker to use than traditional interfaces and also easily discoverable. Further, it allows the specification system to leverage data and schema information to easily rule out impossible joins, thereby accelerating the query process.

3.3.2 *Readable and Pragmatic Layout*

While most gestures such as `PREVIEW` and `UNION` interactions are straightforward from a presentation standpoint, laying out attributes during the `JOIN` interaction faces several challenges. First, due to the textual nature of the information, it needs to be presented in such a way 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.

Research in information visualization has extensively studied layouts such as radial methods [17] for static use cases such as menus, and has also looked into interactive exploration [58] of data. However, the interactive, gesture-driven composition of two tables is a novel and unique setting. To this end, we consider multiple layout options to represent the `JOIN` operation, as shown in Figure 4.

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

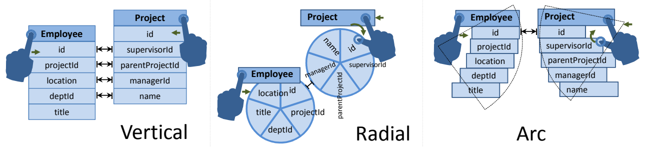


Figure 4: Layout Considerations for the interactive `JOIN`.

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. 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 $2 - -4$ points, 1 or 2 for each table) allows any pair of attributes to be selected as the join predicate.

3.4 Scaling Challenges

We now discuss the impact of scale on the gestural querying paradigm and interface. Due to the iterative nature of query specification, the user can compose *large, complex queries* using a series of intuitive gestures, thanks to the closure of the operations, as described in Section 3.1. Gestures are designed to represent the common case first; subsequent gestures can adapt the query itself, e.g., the default inner equijoins can be changed by swiping the `JOIN` symbol presented up / down to switch out from an equijoin, and right / left to modulate inner / outer participation on each side. Each gesture manipulates the query constructed so far in the workspace. Furthermore, should the user feel the need for *gestural idioms* (i.e. gestures that represent a complex compound query) it is trivial to introduce new gestures into the gesture vocabulary (Section 4). In addition to dealing with larger queries, interacting with *large, complex schema* is a key consideration. The explorative nature of a gestural interface is ideal for navigating larger schema. Gestures described are amenable to the ranking of attributes [40, 16], allowing for operations over relations with a large number of attributes to prioritize gestures towards the most likely queries. Furthermore, techniques such as schema summarization [60] that allow gestures against simplified representations of schema, are ideal future work.

4. QUERY SPECIFICATION SYSTEM

The overall architecture of the query specification system is shown in Figure 5. The user articulates gestures in the gesture query language on a user interface with a dedicated cache for quickly previewing results. These gestures are interpreted by the query specification module, which first translates the gestures into a set of features. The classification step then predicts the most likely query based on the gesture features and the database state. Results are generated using these queries and fed back to the user through the feedback generation module. This section will describe these modules in detail.

4.1 Gesture Recognition as Classification

We model the mapping of gestures (i.e., a collection of time-series of point information) into queries as a classification problem. A gesture is classified as a particular query according to the **proximity** and **compatibility** of the tables involved. 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. Proximity encompasses all of the spatial information about the UI elements, including size, shape, position, and orientation of table and attribute graphical representations along with their velocity and acceleration. Compatibility criteria include schema information like matching field types and similar

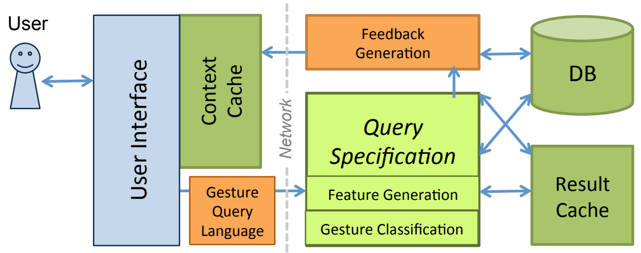


Figure 5: Overall architecture of query specification system.

data using comparisons like join participation histograms, extreme values, intersection in random samples, and total intersection.

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 the finger with which it is associated. 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.

We use a maximum entropy classifier [41, Section 9.2.6] in which we define many "features" of gestures (see Table 1), 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

$$g(q) = \exp\left(\sum_i \lambda_i f_i(q)\right). \tag{1}$$

4.2 Feature Design

The feature functions, $f_i(q)$, for our classifier recognize various constituent parts of gestures and characteristic relationships in the data. New queries can be defined by adding new feature functions and adding new potential queries q to the set of queries. Our two-stage approach takes advantage of an optimization not possible in typical maximum entropy classifiers – features are computed in a specific order and earlier features that return a value of $-\infty$ stop the computation of later features, avoiding unnecessary calculations.

The features used for distinguishing between `JOIN` and `UNION` are shown in Table 1. They include features on the number of touch points involved, the proximity of tables and attributes, and the compatibility of tables and attributes. The features are computed in the order shown in Table 1, with touch number features computed first.

The number of touch points in a gesture immediately rules out certain queries, so the elimination of subsequent unnecessary processing of those queries saves time. Similarly, tables or attributes that are not close enough together (the Close feature) cannot be involved in `JOIN` queries, and tables that are not stacked on top of one another (the Stack feature) cannot be involved in `UNION` queries. By eliminating those possibilities, the computation of unnecessary compatibility features can be eliminated.

Finally, the compatibility features measure whether attributes are the correct type (the Join1 feature) for `JOIN` queries, whether tables have compatible schemas (the U1 feature) for `UNION` queries, and the degree to which two attributes contain similar data (the Join2 feature). Finally, note that within the maximum entropy framework, feature functions can evaluate to discrete values, such as Booleans of either $-\infty$ or 0 (log of 0 and 1), or to continuous values between $-\infty$ and 0 for the Dist and Join2 features.

In the experiments described in Section 5, we compare three different versions of the classifier. The first, called "Proximity",

Table 1: Features used in maximum entropy classifier. Columns are the feature Name, whether it applies to Union and/or Join queries (U/J), the Type of feature (Num: number of tables, Prox: proximity, Comp: compatibility), a description of when it is "On", the entities provided to it as input (1T: one table, 2T: two tables, 2A: two attributes), and its output type (Bool: $\{-\infty, 0\}$, Cont: $(-\infty, 0]$).

Name	U/J	Type	On when	In	Out
Tch1	—	Num	1 simultaneous touch	1T	Bool
Tch2a	—	Num	2 simultaneous touches	1T	Bool
Tch2b	UJ	Num	2 simultaneous touches	2T	Bool
Tch3	J	Num	3 simultaneous touches	2T	Bool
Tch4	J	Num	4 simultaneous touches	2T	Bool
Close	UJ	Prox	Attrs. are close enough	2A	Bool
Dist	UJ	Prox	Negated distance	2A	Cont
Stack	U	Prox	Tables are stacked	2T	Bool
Join1	J	Comp	Attrs. are the same type	2A	Bool
Join2	J	Comp	Data sim. btwn. attrs.	2A	Cont
U1	U	Comp	Tables share schema	2T	Bool
U2	U	Comp	Attr. types match, not ord.	2T	Bool

does not use any of the compatibility features. The second, called "Proximity + Schema" uses the proximity features and compatibility features Join1, U1, and U2, but not Join2. The third version, called "Proximity + Schema + Data" uses all of the proximity and compatibility features.

4.3 Classification Example

To illustrate how the classifier works, consider the example query and gesture described in Section 1.1 with the table layout shown in Figure 1. Before the user begins the gesture, there are no active touches, so all of the TchX feature functions are $-\infty$, making $g(q) = 0$ for all queries q and obviating the need for further processing. When the user touches the headers of the Artist and Album tables, the Tch2b feature function becomes 0 because there are two simultaneous touches on two separate tables while the other TchX feature functions are still $-\infty$. Thus $g(q) = 0$ for queries not involving two touches on two different tables, but processing proceeds on the remaining queries to evaluate $g(q)$. At the beginning of the gesture, all pairs of attributes are far away from each other and the Close feature function is $-\infty$, making $g(q) = 0$ for all queries. Once the tables get close enough together, the Close feature becomes 0 and the Dist features are computed between the six pairs of attributes in the two tables. Because the tables are side-by-side, the Stacked feature is $-\infty$, and only `JOIN` queries have non-zero goodness. After identifying compatible attribute pairs with the Join1 and Join2 feature functions, the closest compatible pair of attributes having the highest goodness is shown as a preview.

4.4 Implementation Details

In addition to the proximity and schema-based compatibility feature functions, our system uses *data similarity* as a feature to aid multi-attribute queries such as joins. This feature is based on a probability distribution over histograms of join participation counts. The motivation for such a measure is an intuition that the user would typically only issue queries that could usefully be displayed on our interface, and thus each value should participate in a small number of results, not too many and not too few. The data compatibility feature, termed Join2, is designed to capture this intuition.

To estimate the data similarity between two attributes, the number of times that each value from one attribute appears in the other is counted and a histogram is constructed from the counts for all of the values. This histogram is then converted to a multinomial distribution and its probability is measured under a dirichlet distribution. In addition, to avoid zero probabilities, we provide a small Laplace prior on the histogram bin counts, so that every bin has 0.01 of a count added to it. This has a large effect on bins with zero counts, but an insignificant effect on bins with non-zero counts. If the histogram bin values are h_i, and the normalized counts are, $x_i = \frac{h_i}{\sum_j h_j}$, then the dirichlet likelihood with parameters α_i is

$$\text{dir}(\mathbf{x}, \alpha) = \frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma(\alpha_i)} \prod_i x_i^{\alpha_i - 1} \tag{2}$$

where $\Gamma(x)$ is the gamma function. This dirichlet likelihood is scaled by a reasonable λ_i to make it comparable to the other goodnesses. In the current experiments, we define the α parameters and the scaling exponent by hand. We define α_i to be

$$\alpha_i = \begin{cases} 1 & 1 \leq i \leq 5 \\ \frac{1}{i-4} & 6 \leq i \leq 20 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In the future, these parameters could be learned from pairs of compatible attributes, although for these experiments we did not have enough pairs to perform such an estimation.

All of the 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 5, these parameters are instead set manually. Learning these tunings from data will allow future quality improvements. We plan to learn these parameters by treating all UI states in a gesture as training instances with the query at which the user eventually arrives as the correct query, much in the same way that search engines learn spell checking [13] and search suggestions from search sessions.

4.5 Context Caching

Note that some of the computations performed in evaluating the feature functions can be shared between different feature functions. For example, the computation of distances between attributes is necessary for the Stack, Dist, and Close features. Similarly, attribute compatibilities are necessary for computing table compatibilities, so computations would be repeated by the U1, U2, and Join1 features. In order to further speed up the system, these computations can be memoized such that they are only computed at most once and only if necessary, with the result cached for subsequent feature functions. The speedup due to this caching is shown in Section 5.4.

5. EXPERIMENTS AND EVALUATION

We perform experiments both from the *user* perspective, evaluating the usability of the system, and from a *systems* perspective, evaluating the performance of the system and sharing insights from the impact of caching strategies employed in the gesture classifier.

In these experiments we test an iPad implementation of the GESTUREQUERY system. The system comprises a frontend written in Javascript and HTML on the iPad and a backend based on Tornado[3] running on a Linux PC. We use publicly available datasets, *Chinook* (11 tables, 64 columns,15K tuples, 1MB)[4] and *TPC-H* (9

[3]`http://www.tornadoweb.org`

[4]`http://chinookdatabase.codeplex.com`

tables, 76 columns, 86K tuples, 15MB)[5] as the test databases[6]. We performed three user studies under procedures approved by our institutional review board[7].

User Study Setup: We adopt a *within-subjects* experimental design since the compared systems were already highly discernible (e.g., CONSOLE-BASED SQL vs. GESTUREQUERY) and catching-on effects were not applicable.

Each experiment was performed over 30 users from the student body of the authors' university, a sample size motivated by prior research in user studies [37, 19, 23]. We report mean and standard errors for all metrics to demonstrate consistency across users, and perform 1-tailed t-tests to establish statistical significance. All studies were carefully designed to avoid bias [37]. Users were recruited from the student body at the university and care was taken to ensure an even distribution of *proficiency* by asking objective post-interview questions about their degree major and prior experience with databases; 15 of the 30 students were confirmed to be proficient at data-related tasks and the other 15 were classified as naive users. For *consistency* each user was given an identical tutorial (read off a written script) on the use of each interface and had a chance to practice each task before their actions were recorded. In order to avoid bias from *learning effects* on the user, the order of the tasks was randomized, as was the order of the systems within each task. *Fatigue effects* were avoided by restricting the length of each experiment and scheduling breaks into the study. *Carryover effects* were avoided by restricting the number of similar tasks, and by counterbalancing (i.e., varying the order of both the systems evaluated and the individual tasks for each user).

Systems and Studies: The interfaces compared were a CONSOLE-BASED SQL (SQLite), the VISUAL QUERY BUILDER of Active Query Builder[8] which represents a state-of-the-art keyboard-and-mouse query interface, and our system, GESTUREQUERY. We performed three studies, *Completion Time*, *Discoverability*, and *Anticipation*, each with their own set of objective and subjective metrics, as described below.

5.1 Completion Time

We first compare the time it takes users to specify queries on three different interfaces. Each student was given six SQL queries to perform on each interface, for example "ALBUM `JOIN` ARTIST ON ALBUM.ARTISTID = ARTIST.ARTISTID".

The *objective metrics* of performance in this study are **completion time**, the total time it takes, on average, to complete each task on each interface, and **interactivity** for `GestureQuery`, the fraction of touches that are classified within a predefined interactive latency threshold. The subjects also completed a survey questionnaire after finishing the study, providing details on their majors of study, relevant coursework and prior experience working with databases. Subjects were also surveyed on *subjective metrics*; **usability** (how easy it was to *use* each system) and **learnability** (how easy it was to *learn* to each system).

Figure 6 shows the average completion time for each action in three different systems. GESTUREQUERY's average completion times are lower and have less variance than both CONSOLE-BASED SQL and VISUAL QUERY BUILDER.

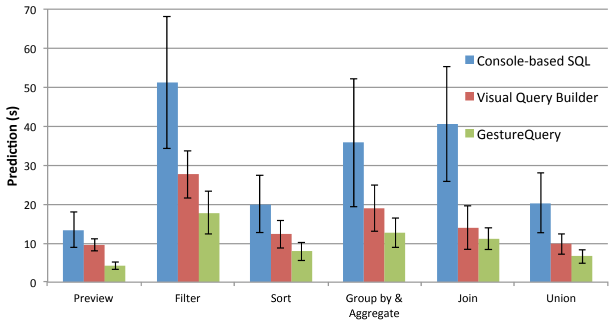


Figure 6: Average completion time for six query types using three different interfaces: Users completed tasks using GESTUREQUERY quicker than the other two interfaces. Error bars show standard error of the mean.

Statistical Significance: In order to determine whether these differences were statistically significant, we performed 1-tailed t-tests on the results. The actions, null hypotheses and p-values are shown in Appendix A. We can clearly see that users construct queries significantly more quickly using GESTUREQUERY than the other two systems, with the significance level under 0.01.

User Survey: Results of the subjective survey questions are shown in Table 2. Each user cast a single vote for each question, and users who selected multiple systems for an answer had their votes divided between those systems. As we can see from the table, most users find GESTUREQUERY easier to use by a large margin except for the `JOIN` action, where VISUAL QUERY BUILDER received 15 votes compared to GESTUREQUERY's 12 for usability. In addition, users think that GESTUREQUERY is easier to learn than console-based SQL and VISUAL QUERY BUILDER except for the `JOIN` operation, where GESTUREQUERY and VISUAL QUERY BUILDER received 14 and 15 votes respectively. In contrast, for `AGGREGATE`, a similarly complex operation, GESTUREQUERY was preferred by most for both usability and learnability.

Drilling down further into the survey results uncovers more insights for the `JOIN` gesture – in terms of *usability*, for proficient users, 9 out of 15 preferred VISUAL QUERY BUILDER and 4 preferred GESTUREQUERY, in contrast to naive users, where GESTUREQUERY received 8 votes and VISUAL QUERY BUILDER received 6. In terms of *learnability*, for proficient users, 9.5 voted for VISUAL QUERY BUILDER and 5.5 voted for GESTUREQUERY (.5 votes for ties), in contrast to naive users, where GESTUREQUERY received 8.5 votes and VISUAL QUERY BUILDER received 5.5. Revisiting individual subjects' interactions reveals that several proficient users prefer to perform `JOIN`s by combining two attributes (e.g. by pinching two attributes together), while naive users have a preference of gesturing compositions at the relation level (by pinching two relations together). Thus, the `JOIN` gesture in `GestureQuery` seems to work better with naive users. This again motivates our ability to customize the gesture vocabulary at the classifier level, discussed in Section 4.

System Performance: Table 3 shows the **interactivity** results, the fraction of gestural inputs (movement of multitouch points on the device) that are classified within 33 ms, a threshold chosen based on the sampling rate of the multitouch sensor, which runs at 30 Hz. Notice that the interactivity results demonstrate a highly fluid interface. There are some gestural inputs that are slower than the interactive threshold. This is caused by the initial loading of schema information into the frontend's Context Cache, which can be improved by prepopulation of relevant information, an area for future work.

[5] `http://www.tpc.org/tpch`

[6] It should be noted that the goal of our experiments is to compare the feasibility of gestural query specification against existing methods; larger, more complex datasets can dealt with using existing summarization methods [50, 60]

[7] OSU IRB #2013B0084

[8] `http://activequerybuilder.com`

Table 2: **Number of users selecting, based on a written survey after the tasks, each system as easiest to use (usability) and learn (learnability) for each query type. Systems are CONSOLE-BASED SQL, VISUAL QUERY BUILDER (VQB), and GESTUREQUERY (GQ – our system). For users claiming equal preference for multiple systems, their votes were split equally amongst the systems.**

Survey	Action	Console	VQB	GQ
Usability	PREVIEW	1	5	**24**
Usability	FILTER	5	1.5	**23.5**
Usability	SORT	0	2	**28**
Usability	AGGREGATE	5	4	**21**
Usability	JOIN	3	**15**	12
Usability	UNION	2	0	**28**
Learnability	PREVIEW	1.66	2.16	**26.16**
Learnability	FILTER	5	4.5	**20.5**
Learnability	SORT	1.33	1.33	**27.33**
Learnability	AGGREGATE	2	10.5	**17.5**
Learnability	JOIN	1	**15**	14
Learnability	UNION	3	0	**27**

Table 3: Interactivity: percentage of touch classifications that take less than 33 ms.

PREVIEW	FILTER	SORT	AGGREGATE	JOIN	UNION
95.1	98.8	97.8	99.6	98.4	98.0

5.2 Discoverability

Another aspect of evaluating ease of use of interaction is to assess how intuitive or *discoverable* the interactions of a system are. Thus, our second study compares VISUAL QUERY BUILDER to GESTUREQUERY in terms of the **discoverability** of the JOIN action, i.e., whether an untrained user is able to intuit how to successfully perform gestural interaction from the interface and its usability affordances [45]. We recruited thirty students for this study, distributed across both proficient and naive users similar to the Completion Time study discussed in the previous section. It should be noted that care was taken to avoid training effects – subjects who had previously performed the Completion Time study were not allowed to perform the Discoverability study. Each subject was provided a task described in natural language, and asked to figure out and complete the query task on each system within 15 minutes. The order of the systems was randomized across subjects to avoid biases from learning effects. The task involved a PREVIEW, FILTER, and JOIN, described to the subjects as answering the question, "What are the titles of the albums created by the artist 'Black Sabbath'?" Before the experiment, we provided a tutorial to each subject on performing the PREVIEW and FILTER actions in both systems, but left it to the user to discover the JOIN action.

Figure 7 shares the results of the discoverability experiment. Twenty-three of the thirty subjects successfully figured out how to complete the task using GESTUREQUERY while only seventeen students were able to complete the task using VISUAL QUERY BUILDER. In addition, for those who did complete the task, the average completion time using VISUAL QUERY BUILDER was more than 19% of that for GESTUREQUERY. If we consider a time-out completion time for the students who don't complete the task as 15 min, the percentage is increased to 31%.

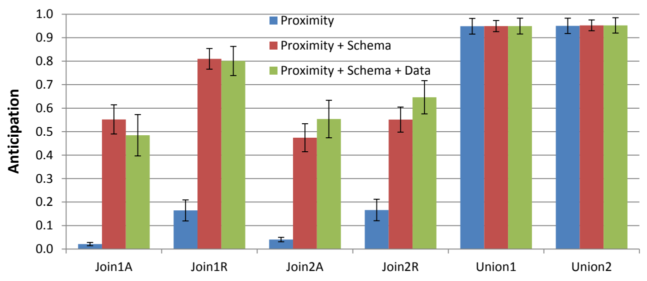


Figure 8: Average anticipation scores for six queries performed by 30 subjects with standard errors. 1.0 represents correct classification at the start of the gesture articulation, and 0 represents an incorrect classification even after completion of the gesture. Information from the schema and data in the database allows our classifier to better predict the intended database query for ambiguous gestures.

5.3 Anticipation

As our third study, we compare the ability of three different classifiers to *anticipate* a user's intent. Thirty of the subjects who participated in the first two studies were recruited to perform two JOIN queries and two UNION queries using GESTUREQUERY. The Joins were performed in two different ways, once by re-arranging the attributes and then dragging the two tables together, and once by adjusting the attribute arc before dragging the two tables together, giving a total of 180 gestures, demonstrating the versatility of the classifier over different gestures. We compared three different versions of our classifier on recordings of these gestures. The first uses only proximity of the UI elements to predict the desired query. The second uses proximity and schema compatibility of attributes and the third uses proximity, schema compatibility, and data compatibility, as described in Section 4. Gestures were recorded until a classifier was able to recognize the correct query. Results are measured as **anticipation**, the fraction of the gesture articulation time remaining after a classifier first recognizes the correct query. This number is 1.0 for gestures that are recognized correctly the moment the articulation process begins, and 0 for gestures that are never recognized.

Results of the anticipation study are presented in Figure 8. Using both proximity and schema compatibility features is better than just proximity in the all JOIN queries. Considering proximity, schema and data compatibility together helps improve anticipation scores for all but the first JOIN, for which data compatibility in fact negatively impacts anticipation scores. Schema and data compatibility are not helpful for the UNION queries because the subjects only perform them on tables that are already fully schema compatible.

Table 4: Cache performance of three versions of classifier on 180 recorded gestures. "Skips prox" and "Prox cache hit" both allow computation to be avoided.

	Prox	Schema	Schema+Data
Computes prox	333,315	81,583	81,583
Skips prox	0	251,732	251,732
Prox cache miss	299,846	74,951	74,951
Prox cache hit	143,750	21,944	21,944

5.4 Optimization Performance

In addition to user-facing numbers, we study the speedup of using context caching, described in Section 4.5, as shown in Table 4 for the three versions of the classifier compared in our experiments. The

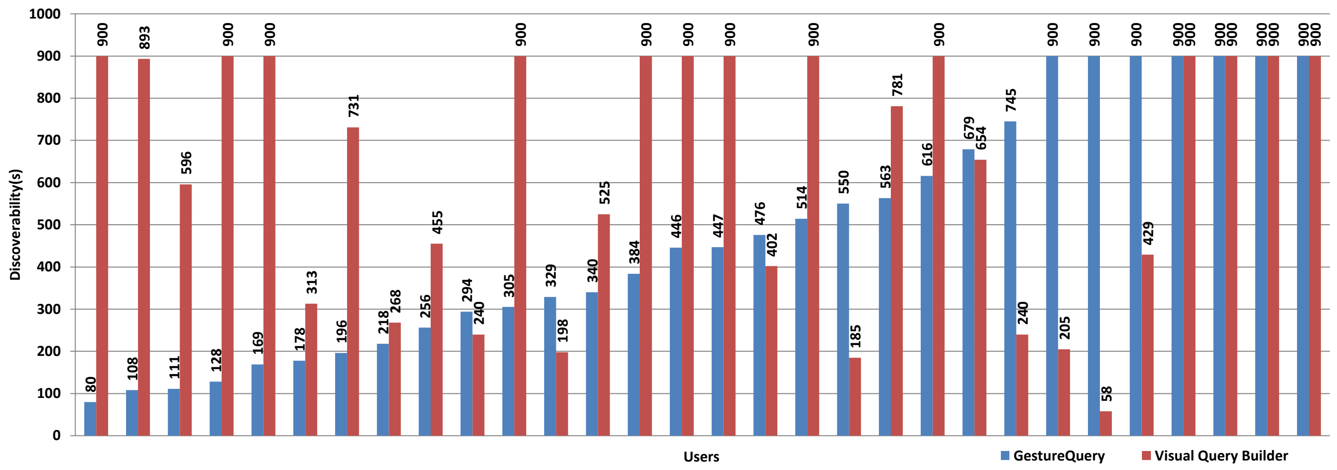


Figure 7: Discoverability: time taken by users to discover how to complete a Join action in GestureQuery and Visual Query Builder, in seconds. Tasks that were not completed within 15 minutes are displayed as 15 minutes.

first optimization is that when the classifier is using compatibility information, it does not need to compute the proximity of incompatible attributes. This leads to a $4\times$ speedup of this computation. The second optimization is to cache the computation of distances between rectangles in the UI. This leads to an additional 23% speedup when the first optimization is active and a 32% speedup when it is not. Note that the “Prox cache hit” and “Prox cache miss” do not add up to the “Computes prox” values because “Computes prox” only includes proximity computations from `JOIN` queries.

6. RELATED WORK

Work towards making databases usable to end users has seen more attention lately [31]. Solutions towards usable data interaction have ranged from innovations in the query paradigm such as natural language, example and output-driven querying [1, 30, 61, 53] to query visualization [22, 15], to user interface innovations in spreadsheet interfaces [7, 6, 38] and autocompletion [43]. Automated form generation [32, 12] assists the process of creating and evolving queries for analytical use cases, while query recommendation [11, 35, 4] efforts using logs have focused on generating the queries themselves. A majority of these query interfaces rely heavily on textual or traditional user interface input which is impractical or infeasible in our context.

Studies on multitouch interfaces [18, 54] have demonstrated the superiority of gestural interaction over mouse-driven WIMP paradigms. Kammer et al. [33] formalize gestures according to the semantics, pragmatics and syntactics of interaction, while Yee [59] discusses concerns regarding direct and indirect gestural interactions. Visual analytics systems such as Tableau [51] and TaP [21] map interactions and gestures from the UI layer to a set of database query templates, without considering the contents of the database itself. Liu et al. [39] have discussed scale challenges in interactive visualization. The visual mining and exploration of datasets [34, 20] has been discussed before, and can be considered ideal applications on top of our database architecture, allowing for both scale and effectiveness of querying.

Interaction with databases using non-keyboard interfaces has received attention, most lately in the vision we have laid out for our system [42, 28]. However, most existing systems map interaction APIs to isolated database queries [25, 47] without any consideration of the query paradigm, query algebra, data or schema of the database [52, 36, 5, 3]. This ad-hoc mapping of gestures, while appealing, does not consider the overall usability of the database querying or its overall effectiveness. In this paper, we posit that a careful consideration of all contexts in the database is essential to building a successful querying mechanism, which we consider to be a key contribution of our work. As demonstrated in our experimental results, this careful consideration pays off in terms of quantifiable improvements in usability.

Probabilistic methods for improving gesture recognition [57] and mapping interaction to actions [14, 48] 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, our classifier performs a two-stage recognition, mapping gesture coordinates to action types, and then using an arc layout and database statistics to successfully identify the exact query.

Work in natural user interfaces [8] has expounded on the use of interfaces in non-keyboard contexts. In the paradigm of direct manipulation, [49] have discussed methods to *directly* interact with data allowing the user to situate the interaction in the same flow as the result output. We are motivated and inspired by these bodies of work. However, **querying databases poses several interaction challenges** even in the context of natural user interfaces and direct manipulation – addressing these challenges is the central focus of our paper. An often-used, and intuitive natural database interface is to map direct manipulation actions to a query algebra [51], thereby offloading the burden of computation to the database. However, in gestural interfaces, several aspects of such a mapping are impractical, motivating our rethinking of the database query paradigm itself.

The use of a feedback loop has been studied in HCI work, especially in assistive input methods [56]. In the context of mixed-initiative user interfaces [26], the idea of prioritization of apparent actions to the end-user is helpful. By modeling the database querying step as a transition from an ambiguous set of queries to an articulate query, our system’s prioritization of feedback during the transition draws from the principle of mixed-initiative interfaces. In database literature, priority scheduling of real-time databases [46] focuses on performance guarantees in multi-tenant contexts, minimizing the number of missed deadlines – work in this area does not consider the query specification stage. For interactive querying, online and approximate query execution [2, 24] focus on the surfacing of answers for explicit queries as they are computed. In our context, the queries themselves are not created yet. Combining dynamic query specification and the development of execution strategies for imprecise, interaction-oriented queries would be an ideal next step.

7. CONCLUSION AND FUTURE WORK

Interacting with databases using gestures is challenging due to the large number of possible database queries. In this paper, we present a novel query specification architecture that allows users to query databases with gestures. We present a novel, well-designed gestural query language to express queries. We build a maximum-entropy classifier-based query recognition system that uses not only multitouch coordinates, but also database state such as schema and data statistics to correctly map multitouch gestures to relational database queries. When compared against conventional coordinate-based gesture mapping, our classifier is more accurate and predictive at identifying intended database queries, and performs well within interactive latency constraints.

We evaluate the efficiency, usability and learnability of our system to users at multiple levels of proficiency in querying databases and our gestural interface. Our detailed user studies demonstrate that our GESTUREQUERY system significantly outperforms current methods on multiple usability metrics for all classes of users.

Going forward, there are multiple challenges that need to be addressed. Some areas of expansion are at the implementation level, such as dealing with textual input in a gesture-oriented interface, which has been well-studied in prior user interface and accessibility literature [56]. Gestural interaction with databases is currently a completely unexplored area of research, and hence is ripe with a wide variety of possible extensions and follow-up problems.

As discussed in Section 3.4, *scaling* the Gestural Query process to more complex queries and to larger and more complex datasets is an immediate next step. For larger datasets, the (possibly approximate) execution of queries within interactive response times requires a fundamental rethinking of the query execution infrastructure. Further, the direct manipulation of complex schema through summarized representations would require extensions to the vocabulary of gestural operations. A detailed study focusing on scaling challenges is ideal follow-up work.

Another area of study is the *feedback generation* used in our system. Intuitively, the feedback presented to the user is meant to aid them in quickly disambiguating the space of possible query intents, while maximizing the entropy of the query results. The interactive summarization of results [50] is significant for constrained display contexts such as mobile devices, and can be leveraged by our system. Further, generating and presenting such feedback *within interactive latencies* is a unique systems challenge, which can be solved using a combination of information visualization methods [10], multi-layer caching techniques and online query execution strategies inspired by Hellerstein et al. [24].

A final area of interest is that of using the gesture information from existing interaction logs to tune the parameters of our classifier, which can then be evaluated to measure the benefits (*completion time* and *anticipation*) and generality (across both users and queries) of user training, allowing us to investigate areas such as *personalization of gestures* and tuning of the gesture prediction itself.

8. ACKNOWLEDGEMENTS

We would like to thank the paper reviewers and Azza Abouzeid for their insights and feedback, which have helped significantly improve the quality of this paper.

9. REFERENCES

[1] A. Abouzied, J. Hellerstein, and A. Silberschatz. DataPlay: Interactive Tweaking and Example-driven Correction of Graphical Database Queries. *UIST*, 2012.

[2] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A Fast Decision Support Systems Using Approximate Query Answers. *VLDB*, 1999.

[3] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. *ICDE*, 2002.

[4] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman. SQL QueRIE Recommendations. *VLDB*, 2010.

[5] S. Amer-Yahia, L. V. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. *SIGMOD*, 2004.

[6] E. Bakke and E. Benson. The Schema-Independent Database UI. *CIDR*, 2011.

[7] E. Bakke, D. Karger, and R. Miller. A Spreadsheet-Based User Interface for Managing Plural Relationships in Structured Data. *CHI*, 2011.

[8] A. Câmara. Natural User Interfaces. *INTERACT*, 2011.

[9] Canalys. Worldwide Smartphone and Client PC Shipment Estimates. 2012.

[10] S. K. Card, J. D. Mackinlay, and B. Schneiderman. *Readings in Information Visualization: Using Vision to Think (Interactive Technologies)*. Morgan Kaufmann, 1999.

[11] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query Recommendations for Interactive Database Exploration. *SSDBM*, 2009.

[12] K. Chen, H. Chen, N. Conway, J. M. Hellerstein, and T. S. Parikh. Usher: Improving Data Quality with Dynamic Forms. *TKDE*, 2011.

[13] S. Cucerzan and E. Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. *EMNLP*, 2004.

[14] S. Damaraju and A. Kerne. Multitouch Gesture Learning and Recognition System. *Tabletops and Interactive Surfaces*, 2008.

[15] J. Danaparamita and W. Gatterbauer. QueryViz: Helping Users Understand SQL Queries and their Patterns. *EDBT*, 2011.

[16] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan. Ordering the Attributes of Query Results. *SIGMOD*, 2006.

[17] G. M. Draper, Y. Livnat, and R. F. Riesenfeld. A Survey of Radial Methods for Information Visualization. *IEEE VCG*, 2009.

[18] S. M. Drucker, D. Fisher, R. Sadana, J. Herron, et al. TouchViz: A Case Study Comparing Two Interfaces for Data Analytics on Tablets. *CHI*, 2013.

[19] L. Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 2003.

[20] M. C. Ferreira de Oliveira and H. Levkowitz. From Visual Data Exploration to Visual Data Mining: A Survey. *IEEE VCG*, 2003.

[21] S. Flöring and T. Hesselmann. TaP: Towards Visual Analytics on Interactive Surfaces. *CoVIS*, 2010.

[22] W. Gatterbauer. Databases will Visualize Queries too. *VLDB*, 2011.

[23] A. Griffin and J. R. Hauser. The Voice of the Customer. *Marketing science*, 1993.

[24] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, et al. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 2000.

[25] S. Hirte, A. Seifert, S. Baumann, D. Klan, and K. Sattler. Data^3 – A Kinect Interface for OLAP using Complex Event Processing. *ICDE*, 2012.
[26] E. Horvitz. Principles of Mixed-Initiative User Interfaces. *CHI*, 1999.
[27] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct Manipulation Interfaces. *Human–Computer Interaction*, 1985.
[28] S. Idreos and E. Liarou. dbTouch: Analytics at your Fingertips. *CIDR*, 2013.
[29] International Data Corporation. Worldwide Quarterly Tablet Tracker. 2013.
[30] Y. Ishikawa, R. Subramanya, and C. Faloutsos. MindReader: Querying databases through multiple examples. *VLDB*, 1998.
[31] H. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making Database Systems Usable. *SIGMOD*, 2007.
[32] M. Jayapandian and H. Jagadish. Automating the Design and Construction of Query Forms. *TKDE*, 2009.
[33] D. Kammer, J. Wojdziak, M. Keck, R. Groh, and S. Taranko. Towards a Formalization of Multi-touch Gestures. *ITS*, 2010.
[34] D. A. Keim. Visual Exploration of Large Data Sets. *CACM*, 2001.
[35] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-Aware Autocompletion for SQL. *VLDB*, 2010.
[36] B. Kimelfeld and Y. Sagiv. Finding and Approximating Top-k Answers in Keyword Proximity Search. *PODS*, 2006.
[37] J. Lazar, J. H. Feng, and H. Hochheiser. *Research Methods in Human-Computer Interaction*. Wiley. com, 2010.
[38] B. Liu and H. Jagadish. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. *ICDE*, 2009.
[39] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time Visual Querying of Big Data. *EuroVis*, 2013.
[40] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing Out in a Crowd: Selecting Attributes for Maximum Visibility. *ICDE*, 2008.
[41] K. P. Murphy. *Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series)*. MIT Press, 2012.
[42] A. Nandi. Querying Without Keyboards. *CIDR*, 2013.
[43] A. Nandi and H. Jagadish. Assisted Querying using Instant-response Interfaces. *SIGMOD*, 2007.
[44] A. Nandi and H. Jagadish. Guided Interaction: Rethinking the Query-Result Paradigm. *VLDB*, 2011.
[45] D. A. Norman. Affordance, Conventions, and Design. *Interactions*, 1999.
[46] H. Pang, M. J. Carey, and M. Livny. Multiclass Query Scheduling in Real-Time Database Systems. *TKDE*, 1995.
[47] S. Patney et al. SQL Server Kinection. *PASS*, 2011.
[48] J. Schwarz, J. Mankoff, and S. Hudson. Monte Carlo Methods for Managing Interactive State, Action and Feedback Under Uncertainty. *UIST*, 2011.
[49] B. Shneiderman, C. Williamson, and C. Ahlberg. Dynamic Queries: Database Searching by Direct Manipulation. *CHI*, 1992.
[50] M. Singh, A. Nandi, and H. Jagadish. Skimmer: Rapid Scrolling of Relational Query Results. *SIGMOD*, 2012.
[51] C. Stolte. Visual Interfaces to Data. *SIGMOD*, 2010.
[52] A. Termehchy and M. Winslett. Using Structural Information in XML Keyword Search Effectively. *TODS*, 2011.
[53] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by Output. *SIGMOD*, 2009.
[54] B. Ullmer and H. Ishii. Emerging Frameworks for Tangible User Interfaces. *IBM Sys. Journal*, 2000.
[55] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On Query Result Diversification. *ICDE*, 2011.
[56] D. J. Ward, A. F. Blackwell, and D. J. MacKay. Dasher – a Data Entry Interface Using Continuous Gestures and Language Models. *UIST*, 2000.
[57] D. Weir, S. Rogers, R. Murray-Smith, and M. Löchtefeld. A User-Specific Machine Learning Approach for Improving Touch Accuracy on Mobile Devices. *UIST*, 2012.
[58] G. J. Wills. NicheWorks – Interactive Visualization of Very Large Graphs. *Computational and Graphical Statistics*, 1999.
[59] W. Yee. Potential Limitations of Multi-touch Gesture Vocabulary: Differentiation, Adoption, Fatigue. *Human-Computer Interaction: Novel Interaction Methods and Techniques*, 2009.
[60] C. Yu and H. Jagadish. Schema Summarization. *VLDB*, 2006.
[61] M. Zloof. Query by Example. *NCCE*, 1975.

APPENDIX

$anticipation_c$ represents the anticipation time of the console-based SQL, $anticipation_v$ represents the anticipation time of the visual query builder and $anticipation_g$ represents the anticipation time of our system.

A. T-TEST RESULT

Action	Null hypothesis	P-value
Preview	$anticipation_c \leq anticipation_g$	1.93E-12
Preview	$anticipation_v \leq anticipation_g$	2.20E-16
Filter	$anticipation_c \leq anticipation_g$	1.14E-12
Filter	$anticipation_v \leq anticipation_g$	6.79E-09
Sort	$anticipation_c \leq anticipation_g$	4.50E-10
Sort	$anticipation_v \leq anticipation_g$	1.29E-05
Group by & Aggregate	$anticipation_c \leq anticipation_g$	1.66E-09
Group by & Aggregate	$anticipation_v \leq anticipation_g$	1.26E-05
Join	$anticipation_c \leq anticipation_g$	1.54E-12
Join	$anticipation_v \leq anticipation_g$	0.007703
Union	$anticipation_c \leq anticipation_g$	2.76E-10
Union	$anticipation_v \leq anticipation_g$	8.90E-06