Querying Without Keyboards

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

ABSTRACT

Computing devices that use non-traditional methods to interact with data are becoming more popular than those that use keyboard-based interaction. Applications and query interfaces for such devices pose a fundamentally different set of workloads on the underlying databases. We characterize these differences and show how today's query representation and evaluation techniques are not well-suited to these new non-keyboard interaction modalities. We propose a new database architecture to address these issues, and demonstrate that it can be built using already existing components.

1. INTRODUCTION

The proliferation of next-generation computing devices such as tablets, smartphones, gesture-based systems such as the Kinect, and eye-tracking-based systems such as Google Glass have ushered us into a new age of end-user access to data. In 2011, over 550 million smartphones and tablets were sold – 1.5 times that of the number of desktops, laptops, and netbooks combined [5] for the same period. In the last quarter of that year, this ratio jumped to 1.9 times. Given this trend, 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.

Users are also performing a immensely wide variety of fairly advanced tasks through these interfaces: from making flight reservations (e.g., a third of frequent flyers use tablets and smartphones to make reservations[2]) to big data analytics [21]. The plethora of new applications built for such environments also introduces a wide vocabulary of interaction paradigms, such as direct manipulation [22], multi-touch, gestures, and multi-user input. Applications are increasingly communicating more data to the user, featuring visualizations such as sparklines and heatmaps as core parts of the user interface, and allowing for fluid interactions with large amounts of data both of which involve an increasing reliance on the underlying data layer.

The current generation of data interfaces – forms, reporting tools, and query workflows – are all powered by backends that are built around the *Query→Result* paradigm. For example, applications extract all information needs from a user, prepare a query for the database using SQL directly or through an ORM, and then communicate the query to a database which executes the query and returns the results. The design of such interfaces can be traced back to text-based console input, involving a back-and-forth dialog with the computer. In traditional interfaces, there is no activity *during* the query construction process. Interactive tools of today that use different interaction paradigms, such as direct manipulation, are being shoehorned into typical access patterns built for keyboards and mice, which can no longer be assumed as the sole way to access data.

New paradigms to data interaction are orthogonal to the user's skill levels. Graphical user interfaces have typically been perceived as being an "easier" but less powerful approach to interacting with databases. In contrast, our efforts are independent of user proficiency: we see the new interaction patterns as an opportunity to efficiently issue more advanced queries than possible with a purely text-oriented approach.

2. CHALLENGES

A primary challenge with modern data-driven applications is the change in user expectations of *query latency*. Due to the popularity of web services, querying is expected to be near-instantaneous. In the case of direct manipulation, a paradigm adopted by most non-keyboard interfaces, fluid interactivity relies on instant feedback: calls to the underlying data layer are considered blocking factors. The current model of database querying, however, does not place a bound on execution time of queries, possibly breaking the interaction flow due to a long-running query.

Second, modern application layers, especially those from non-traditional interfaces, issue a *significantly different workload* than what is currently expected by the database. As an example, consider the browsing of result sets. In traditional scenarios, the result set is materialized and stored in a (temporary) relation. The user is presented with a paginated tabular display, showing the first n tuples of the result set using extensions such as TOP or $LIMIT$. The user then clicks the "Next" button to jump to the next n results and "Previous" to go back, which then trigger additional SQL queries, lazily loading pages as they come into view. Adaptations of this interfaces for mouse-based interfaces overload the scrolling mechanism, with scroll up and scroll down denoting the previously mentioned actions.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6–9, 2013, Asilomar, California, USA.

The advent of accelerated scrolling for touch-driven interfaces poses an interesting challenge to this setup. A simple swipe can trigger an accelerated scroll on a relation, effectively issuing a large number of queries (one for each "page" of information) to the database backend. Worse, continued swiping accelerates the scrolling speed, issuing a superlinear number of queries, overloading the query queue. Ironically, due to the high speed of scrolling, most of these result pages are ignored by the user. The lack of responsiveness on an interface due to an unresponsive backend will typically cause the user to attempt more interactions (and hence triggering more queries), exacerbating the problem.

The scheduling of queries from a scrolling interface is an interesting challenge: Since a user will typically be interested in the current page, do we kill all old queries and prioritize all new queries? Further, due to the ephemerality of some results, can we get away with approximate, partial answers? [23] Clearly, a cohesive answer to these questions is a highly desirable goal.

A third challenge is the *inherent exploratory nature* of non-keyboard interfaces. As we will see in Section 3.1, feedback on a prospective join is shown for the most proximal pair of attributes in a multi-touch interface. In contrast to traditional textual input where queries are executed when "Enter" is pressed, each finger movement could possibly trigger a different join (in the worst case, all possible attribute combinations could be valid joins). Each join is executed and then post-processed to display a summary of the results, such as participation cardinality, and a (post-processed) preview of the data. Executing such queries to completion would be prohibitively expensive without careful optimization. For significantly large datasets, providing perceivably instantaneous feedback (i.e. in the 100ms range) would not be possible. As is the case with most exploratory interfaces, a majority of the queries are simply "what-if" questions posed by the user; who is simply trying to articulate an explicit query by trial-and-error – a majority of the complete materialized results will be ignored by the user anyway. Thus, a method to prioritize information that aids in fulfilling the user's final information need, would reduce unnecessary exploration.

The fourth challenge is that of providing feedback for non-keyboard interaction. Unlike textual interfaces where we can use the input directly, feedback will need to be generated using a combination of interface state/context (e.g. "which items have been touched so far?"), gestural cues (e.g. "can we recognize that the user made a pinch-out gesture on a resultset?"), and positional information (e.g. "what is the distance between dragged items X and Y, and what are their velocities and direction?"). Providing feedback to users *during* such input or gesture is critical. Many of these inputs (such as position), unlike text, can change rapidly over time, which the feedback generation mechanisms should support.

Outline: In the following section we describe the *QWiK* system, where we begin with an example user interaction. We describe our Query Model and System Architecture in Sections 3.3 and 3.2, followed by a walkthrough of the example query through our system. We conclude with Related Work in Section 4.

3. A QWIK DATABASE

Based on the motivation and challenges, we now propose the *QWiK* (Querying Without Keyboards) database system. This system is designed to serve touch, gesture, and other natural user interfaces which have workloads similar to our motivating example, described below. The user interface (*one* possible interface – others can be built on top of our system) allows interaction with databases using multi-touch gestures. While the interface is capable of performing more complex compositions, for this paper, we focus on the *interactive join* gesture, which allows users to perform a join on two relations.

3.1 Motivating Example

Consider an equijoin between two relations **Employee** (E) and **Project** (P), as shown in Figure 1. Based on the attributes, there are four valid (based on key relationships in the schema) joins between the two relations:$E.projectId = P.id$, $E.projectId = P.parentProjectId$, $E.id = P.supervisorId$ and $E.id = P.managerId$. Each join yields a different, valid result with different participation and tuple cardinalities.

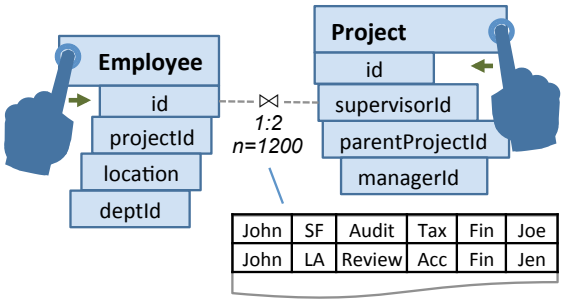

Figure 1: User Interface for an interactive join operation, with feedback on prospective joins. Current systems encounter several challenges when catering to workloads from such interfaces.

Interface Behavior: As the user brings the tiles for the two relations close to each other, the interface detects the movement and begins suggesting possible query actions. It arranges the attributes such that they are amenable to joining together. Further, the most likely pair of attributes to be joined are depicted using a dashed connector. Cardinality information of the participating tuples is presented, and a preview of the candidate results are shown as well. If this is the intended choice, the user simply moves the tiles in closer till the attributes touch, confirming the join. If this is not the intended join, the user moves tiles towards the intended attribute pair. This causes the connector to switch to the next attribute combination. All feedback from the system is fully interactive – the system reacts to each pixel movement of the user, responding instantaneously.

This feedback allows the user to compose relations in a fluid and intuitive manner. All user movements are responded to with instantaneous insights to the result of the possible query, thereby guiding the user towards constructing the query originally intended by the user, with no dependency on prior knowledge of the schema, data or even the query language. While our current example schema is fairly articulate in its naming and structure, the user would benefit a lot more from this interface given a database with no predefined key constraints or well-named relations / attributes.

3.2 Query Model

We now introduce our query model, that builds upon the relational database model.

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 an *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 session. Due to the directly manipulable nature of the interface, the system maintains a list of all query intent transitions, and all recent intermediate results and feedback. 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 we will discuss 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 infer 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? To do this, we attempt to maximize the entropy of the feedback, in the hope that this will cause the user to polarize opinions on a certain type of query, keeping in mind the size of the feedback and the probability of query:

$$Feedback = \arg\max_{r \subset R_U} \left(\frac{1}{|r|} \cdot H(r) \cdot \sum_{q \in Q(r)} P(q) \right)$$

Where H is the entropy of a resultset. R_U is the combined set of results from all queries in the intent space. $Q(r)$ is the set of queries in the intent space whose results contain r. $P(q)$ is the probability of query q in the query intent.

While user input such as touch movement may change rapidly, it causes a slower change in the intent space, most likely just a re-weighting of the probability distribution. Results are generated in an online fashion, allowing for time-bounded computation of feedback. The feedback loop runs incrementally, reusing computations (stored in the context) from the previous iteration. This allows our query model to successfully tackle all challenges discussed in Section 2.

3.3 System Architecture

We now describe the various components of the QWiK system, as illustrated in Figure 2. The user interacts with the user interface, that connects over the network to the QWiK backend. The query context is maintained both at the frontend and the backend for performance reasons. The Gesture Mapping module maps interactions in the user interface to corresponding database query templates. It should be noted that while the user interface and gesture mapping components are a critical part of the end-to-end system, they are orthogonal to the core QWiK architecture, and can be replaced with other interfaces. In the backend, the system is divided into two parts. The Intent Interpretation module uses the candidate queries from the incoming gestures, the query context and the database to infer the query intent space. The Feedback Generation module then generates an optimized set of insights to present to the user, by executing queries, leveraging index and catalog information from the database, and considering intent history from the query context. Generation is done in an online manner, returning a "best-so-far" feedback in bounded time. Resultsets typically get reused through a query intent transition, and can be improved across feedback computations.

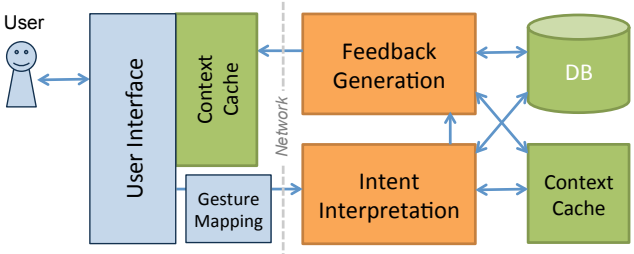

Figure 2: QWiK System Architecture that can be used to power non-keyboard interfaces, e.g. Figure 1. *Feedback Generation* and *Intent Interpretation* tackle database challenges critical to our use case.

Query Walkthrough: To explain the architecture, we provide a walkthrough our interactive join example. The user interface is implemented as an multi-touch application, which connects over the network to the QWiK backend. The join action, when looked at carefully, is a combination of three gestures: touching the first relation, then the second, followed by dragging the relations towards each other till the participating attributes are in contact.

The user first selects a tile representing a single relation on the user interface by touching it on the screen. This is recognized as the "select relation" gesture on the `EMPLOYEE` relation, based on the nature of the tile, and the touch action and returns the filled query template `SELECT * FROM [relation:EMPLOYEE]`, where `EMPLOYEE` was filled in by the module. The Intent Interpretation module then observes an empty context cache. Based on the current gesture, the intent space is clearly a single query, with probability 1.0. The work of the feedback module is thus simple: it simply returns a preview of the relation by issuing the following queries: `SELECT * FROM` *`EMPLOYEE`* `LIMIT 10` and `SELECT COUNT(*)`

FROM *EMPLOYEE*. Both results are stored in the query cache along with the explicit query. The results are then sent to the frontend, which populates the cache in the frontend, and updates the display by "opening" the relation's tile into the set of attributes, as shown in Figure 1. The preview area shows sample tuples and cardinality of the relation.

The same action takes place upon touching the second relation, PROJECT, with the difference that the context contained the previous query. However, due to the fact that the prior touch event was not in motion (i.e. the user was not dragging it), and hence not the primary focus, the probability of that query was set to zero by the interface, thus constraining the query intent space, and allowing the feedback module to return only results from the PROJECT table.

The third action is the dragging motion. Once the tiles are within a certain distance, the *join* gesture is recognized, and the SELECT * FROM [relation:EMPLOYEE] JOIN [relation:PROJECT] ON [attribX = attribY] is returned. The Intent Interpretation module uses the schema information in the query context (populated from the prior queries) to rewrite this as the set of four possible joins, listed in Section 3.1. The normalized proximity between corresponding attribute pairs is used to assign probabilities to each query in the intent. The feedback module now uses the database catalog information and indexes to estimate participation cardinalities for each possible join, and attempts to materialize preview tuples. The feedback (however much has materialized in the bounded time) is then sent to the frontend. At the frontend, the feedback information for the most proximal pair is displayed. As the user moves the relations to align the correct attributes close to each other, the probability of the intended pair is increased, and hence the feedback loop shows more tuples from that join in the preview area (tuples are postprocessed by joining against fact relations for readability). Since the dragging motion is merely a reweighting of probabilities in the intent space, most feedback computations are cached. Upon deciding which attribute pair to pick, the user then brings the attributes in contact, confirming the join at the user interface level. This updates the probability of that join to 1.0, triggering a full materialization by the feedback module, and the corresponding postprocessing and transfer of summarized results to the frontend.

4. RELATED WORK

Work in the HCI community, specifically with natural user interfaces [4] and direct manipulation [22] have discussed methods to interact with data in non-keyboard contexts. While we are motivated and inspired by these bodies of work, we recognize that **the database layer that powers such interfaces poses several challenges, which we address in this paper**. A common current solution is to map direct manipulation actions to a query algebra [24] offloading the burden of computation on the database. There are several aspects in which such a mapping is impractical, motivating our rethinking of the database layer itself. Efforts in using gestures to interact with databases [10, 20] are fairly primitive and simply map gestures to isolated query commands, with no consideration of database query context or feedback being used as part of the gesturing process.

Efforts towards making databases more usable [12] range from automatic generation and evolution of query forms [7, 13] to modeling databases as spreadsheets [3, 16], or as autocompletion input [17]. These efforts rely heavily on textual input; which is infeasible in our context.

Example-based query interfaces [25] and the recommendation of queries using prior logs [6, 15] has also been explored. Exploration and mining of datasets using visual methods [8, 14] can be considered applications on top of our database architecture. We have previously proposed design principles to expose insights from the database in a guided manner [18]. Such ideas are substantially more important in non-keyboard query environments where the cost of articulating the query is high.

The prioritization of insights and their inclusion in the feedback loop is a common problem in HCI and has been discussed at length in the context of mixed-initiative user interfaces [11]. In databases, priority scheduling in real-time databases [19] focuses on utilizing an objective that minimizes the number of missed deadlines. Research in online and approximate query execution [1, 9] focuses on early surfacing of answers for explicit queries. We consider these ideas foundational work for our vision. Developing execution strategies for imprecise, interaction-oriented workloads is a promising area of future research.

References

[1] S. Acharya et al. Aqua: Decision Support Systems using Approximate Query Answers. In *VLDB*, 1999.
[2] Amadeus IT Group. How Mobile Will Transform the Future of Air Travel. 2011.
[3] E. Bakke. Schema-independent DB UI. *CIDR*, 2011.
[4] A. Câmara. Natural User Interfaces. In *IFIP*, 2011.
[5] Canalys. Worldwide Smartphone and Client PC Shipment Estimates. 2012.
[6] G. Chatzopoulou et al. Query Recommendations for Interactive Database Exploration. In *SSDBM*, 2009.
[7] K. Chen, H. Chen, et al. Usher: Improving Data Quality With Dynamic Forms. In *ICDE*, 2010.
[8] F. de Oliveira et al. From Visual Data Exploration to Visual Data Mining. *IEEE VCG*, 2003.
[9] J. Hellerstein, M. Franklin, et al. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 2000.
[10] S. Hirte, A. Seifert, S. Baumann, and D. Klan. Data^3 – A Kinect Interface for OLAP. In *ICDE*, 2012.
[11] E. Horvitz. Principles of Mixed-Initiative User Interfaces. In *CHI*, 1999.
[12] H. Jagadish et al. Making Database Systems Usable. In *SIGMOD*, 2007.
[13] M. Jayapandian and H. Jagadish. Automating the Design and Construction of Query Forms. *TKDE*, 2009.
[14] D. Keim. Visual Exploration of Datasets. *CACM*, 2001.
[15] N. Khoussainova et al. SnipSuggest: Context-aware Autocompletion for SQL. *VLDB*, 2010.
[16] B. Liu and H. Jagadish. A Spreadsheet Algebra for a Direct Manipulation Interface. In *ICDE*, 2009.
[17] A. Nandi and H. Jagadish. Assisted Querying Using Instant-Response Interfaces. In *SIGMOD*, 2007.
[18] A. Nandi and H. Jagadish. Guided Interaction: Rethinking the Query-Result Paradigm. *VLDB*, 2011.
[19] H. Pang, M. Carey, and M. Livny. Multiclass Query Scheduling in RTDBMS. *TKDE*, 1995.
[20] S. Patney et al. SQL Server Kinection. *PASS*, 2011.
[21] Roambi Inc. Roambi Analytics.
[22] B. Shneiderman et al. Dynamic Queries: Database Searching by Direct Manipulation. In *CHI*, 1992.
[23] M. Singh, A. Nandi, and H. Jagadish. Skimmer: Rapid Scrolling of Relational Query Results. *SIGMOD*, 2012.
[24] C. Stolte. Visual Interfaces to Data. In *SIGMOD*, 2010.
[25] M. Zloof. Query by Example. In *NCCE*, 1975.