ARQuery: Hallucinating Analytics over Real-World Data using Augmented Reality

Codi Burley
Computer Science & Engineering
The Ohio State University
burley.66@osu.edu

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

ABSTRACT

In addition to the virtual, there is a vast amount of data present in the real world. Given recent advances in computer vision, augmented reality, and cloud services, we are faced with a tremendous opportunity to augment the structured data around end-users with insights. Coinciding with these trends, the number of data-rich end-user activities is also rapidly increasing. Thus, it is useful to investigate the process of data exploration and analysis in augmented and mixed reality settings. In this paper, we describe *ARQuery*, a query platform that utilizes augmented reality to enable querying over real-world data. We provide an interaction and visualization grammar that is designed to augment real-world data, and a performant framework that enables query exploration in real-time. Our studies show that ARQuery provides a fluid, low-latency query experience for the end-user that is significantly faster than traditional approaches.

1. INTRODUCTION

Our lives are increasingly swimming in a deluge of data. Beyond the data accessible in the virtual world, we are immersed with structured data in the *real-world* as well: from paper-printed restaurant menus and nutrition labels, to digital signage-based flight schedules at airports and bus stops. Unlike data that we own, real-world data is truly ad-hoc: we deal with it all the time, but cannot predict exactly when or how we will need to interact with it. Furthermore, since the displayed data (paper, or digital) is not inside a virtual environment that we control; quickly "querying" such data is typically done manually, e.g., looking for *the least calorific non-allergenic appetizers under $20* or determining the *total expenses per category* in a large itemized invoice. While these can be trivially represented as analytical queries, our real-life experiences still involve poring through such data and performing mental calculations.

At the same time, over the past few decades, *augmented reality* – a technology to overlay a live view of the physical world with digital information – has gone from a science fiction concept to a commodity, consumer-grade technology used by millions [1]. Augmented Reality (AR) has found a variety of specialized use cases beyond just gaming [3]: areas such as education [28], medicine [12], and emergency services [23] have each enabled a completely new form of digital interaction using camera-based AR devices. This mode of interaction is expected to rise sharply due to three complementary trends: recent advancements in computer vision & augmented reality research, population-scale availability of camera-enabled hardware, and affordability of cloud-backed edge devices.

There have been several impressive improvements in computer vision research [22] recently, to the point where fairly advanced techniques are now available as consumer-grade hardware and software, and also reliable building blocks for other research. This has triggered a wave of high-quality services (Google Vision, Amazon Rekognition, Clarifai) and open source models/libraries (Tensorflow, Caffe, OpenCV) that can be considered commodity. Furthermore, augmented reality (*AR*) wearable devices, such as Google Glass, Microsoft HoloLens, and Magic Leap have become available. These devices continually capture and process image and video data and provide pertinent feedback, i.e., *augmentation*, through an overlay display. These devices have inspired and unlocked a variety of "camera-first" interaction modalities, where the camera is often the primary mode of capture and input – and this paradigm is transferring over to smartphones and tablets as well. AR applications such as Snapchat, Google Lens, and Amazon Shopping are bringing a completely new and natural mode of interaction to consumer-grade smartphones and tablets [14].

1.1 The need for a querying framework in augmented reality

Coinciding with these device trends, the number of *data-rich applications*: end-user activities that are backed by large amounts of data is also rapidly increasing. For example, a simple restaurant lookup on Google Search and Google Maps is now augmented with wait times, popular hours, and visit duration, aggregated from population-scale user location history logs [10]. By considering a user's AR view as a *queried view* on a data store, this data-rich paradigm brings about a unique opportunity to build compelling querying experiences for end-users. Just as touchscreen interfaces triggered a body of research [18, 15] and products [6] in touch-based querying of structured data, we expect augmented reality to also spur an entirely new body of work in data analysis and exploration.

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

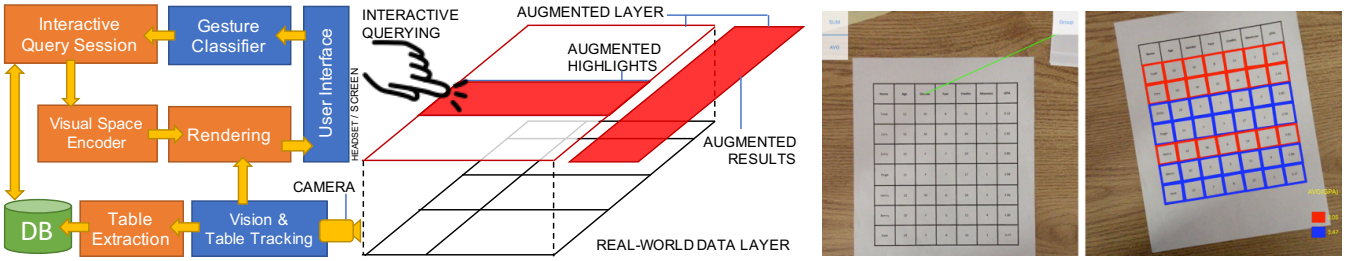

Figure 1: Overall architecture and layers of ARQuery. In this query, a `GROUP BY` aggregation is performed on a sheet of paper by swiping right on an attribute, yielding a divergent color coding highlights to represent groups, and the aggregates presented as an augmented result legend in the bottom right.

In all data-rich settings – whether it is consumer, industrial, enterprise, or scientific audiences – there is tremendous opportunity to query the real-world structured data around the user, and augment it with cloud-based data sources.

1.2 Motivating Example and Challenges

Christina is frequent flyer who learns that her United Airlines flight to Monterey *(MRY)* was cancelled to bad weather, and that she has been rebooked for the next day. Since arriving a day late would mean she would miss her own keynote talk at CIDR, she decides to purchase a ticket on a different airline. While she has a bevy of airline apps on her phone, the changing weather is triggering new delays – and the most up-to-date information about departing flights is available in front of her on a large and unwieldy terminal display, sorted by departure time. Despite time being of the essence, Christina has to manually pore through each display entry to see if it is a candidate.

Figure 2: Motivating Example: Querying on airport schedules.

With ARQuery, Christina simply views the airport display through her iPad's camera, and tap-holds on *MRY* to filter out from her view all flights that do not go to *MRY*. Then, she swipes the *"Boarding Time"* header to order the remainder flights by boarding time. She identifies an American Airlines flight boarding in 10 minutes, and walks over to the airline's customer desk to make the purchase. By quickly performing a `SELECT * FROM flights WHERE departing_to='Monterey' ORDER BY boarding_time` query in augmented reality, Christina is able to successfully make it to CIDR to deliver her keynote.

ARQuery enables *any* source of structured data – either digital or paper – to be queried using a commodity device within seconds. All alternatives such as apps and manual interactions would take much longer, and in this case – possibly lead to unwanted outcomes such as missed flights. By providing a fast, real-time interaction experience, the user is provided an experience akin to that of "hallucinating" a visual data exploration session in the real-world. While this example used ARQuery through a tablet, we envision headsets such as Microsoft Hololens to become consumer viable over time, allowing a headset-based implementation of ARQuery to be used in general purpose data exploration as well.

As evidenced by the popularity of AR-based applications such as Pokemon Go, there is widespread availability of AR-capable hardware and user interest in this new mode of interaction. Similarly, optical character recognition has reached unprecedented quality lately, and is available in consumer smartphone apps. However, from a querying standpoint, several core challenges exist, which we address with ARQuery. First, unlike objects on a screen, data in the real-world(e.g., paper) persists throughout the query session, hence our system needs to *augment existing data* instead of replacing it. Second, all queries are performed using direct manipulation of real-world data, meaning that a visual grammar needs to be designed for interaction with such data and augmentations. Finally, from a performance standpoint, all interactions need to be at a perceptibly instantaneous framerate, necessitating fast table inference, querying, and result-rendering across the camera's video stream.

1.3 Contributions

Based on these challenges and opportunities, we present *ARQuery*, a platform for querying in augmented reality. It provides

- a data model with result spaces fit for visually representing result sets in AR,
- an interaction and visualization grammar to query structured data in the real-world using a series of gestures, and
- a performant framework that enables such interactions over datasets in real time.

Our studies show that ARQuery fares extremely well both in terms of real-time performance, and allowing users to perform queries 3– 10× faster than traditional approaches.

2. RELATED WORK

ARQuery builds upon a rich body of work in using direct manipulation to perform ad-hoc querying of structured data, ranging from the mouse-based Polaris [21] (available commercially as Tableau) to pen, gesture, or touch-based systems, such as DBTouch [15], Kinetica [20], Vizdom [8], and GestureQuery [18]. While cloud-backed exploration of augmented reality has been quite popular with exploring physical environments through games such as *Pokemon Go* or shopping interactions [24], our work focuses on structured data in real-world settings, instead of just physical objects. From a real-world data perspective, our work aligns with Shreddr [7], which investigated the digitization and management of paper-based data. It should be noted that our work considers digitization an orthogonal black-box dependency – recent advances in this avenue of computer vision [11] have drastically improved in terms of accuracy and performance and are used in everyday life [9], to the point where we have been successfully able to use commodity OCR APIs with near-perfect outcomes. From an interaction grammar perspective, ARQuery maps interactions to the query operations in relational algebra, reminiscent of VizQL [13] and GestureDB [19], but is limited by *augmentation*: instead of showing a sequence of visual query transformations [17], all query results are represented as "hallucination-style" annotations over real-world data by moving into a visual encoding over relations.

3. DATA MODEL AND RESULT SPACES

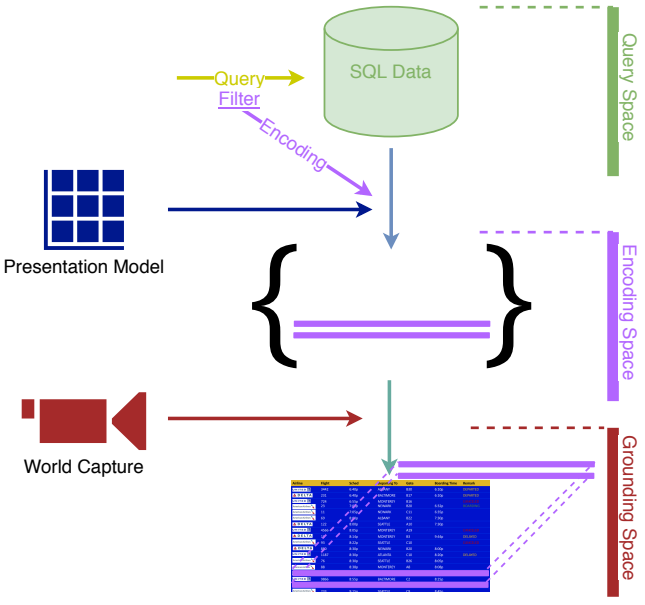

Figure 3: An overview of the data model used by ARQuery, where multiple result spaces are used to visually encode result sets in AR.

ARQuery works on data that is present in the real-world in a tabular form. We use the SQL data model in our interpretation of tabular data. Our system currently supports only read queries (writing to the real-world is considered future work). Similar to [17], we model our query session as a series of operations that successively modify the result set. The space of queries we cover is defined by all possible compositions of the operations we allow: selection, projection, group, aggregate and ordering. Binary operators such as union and join are not yet supported in the system, but can be approximated by concatenating or juxtaposing multiple real-world tables (see Future Work section for a discussion). We do, however, present a method for encoding joins in AR using our visual querying grammar (see section 5.3). Note that this model alone does not provide a means for visually representing query results to the user in AR. To address this problem we introduce two new spaces, the *encoding space* and the *grounding space*. These provide visual representations of result sets from the query space, where the underlying relational data model is contained. Ultimately, these additions allow us to convey query results to the user in a real time AR environment. The encoding space uses the presentation model obtained from the real world and queries on the underlying data to produce visual artifacts that encode result sets. Results from the encoding space are transformed onto the real world in what we will call the grounding space. These grounded visual artifacts serve as the final result set representation in ARQuery. This mapping between spaces is outlined in Figure 3. These new spaces, the necessity for them, and their interactions are described in more detail in the following sections.

3.1 The Encoding Space

A core challenge with mixed reality environments is the presence of reality itself – unlike completely virtual environments, it is difficult to erase objects from the real-world. For example, *sorting* a table in the real-world would imply possibly rearranging all its cells. Since this is not possible, we utilize *color* as a way to encode *order position*. Thus, all changes to the result set need to be encoded in an *additive* manner to the original real-world data table. These additive encodings are often not possible in the relational space (as shown with the sorting example), so it is necessary to move to an encoding space for appropriate representations.

The encoding space is parameterized by a mapping from the data model to a presentation model and encoding functions. The presentation model we use for encoding onto a real world table is defined by the components that make up a table and their relative positions in a two dimensional coordinate system defined by the bounding box of the table. The components that can be referenced are individual cells, rows, columns, and headers of columns. They are stored in a 2-D array that is oriented as the query session table is in reality, allowing for easy access.

The encoding functions are what produce the actual visual artifacts, utilizing the mapping to the presentation model and various other parameters. These visual artifacts require the presentation model's relative coordinate system for positioning, but are not limited to the dimension of it. For example, we could encode order-by attributes with the height of 3-D histogram bars that are placed on top of table rows from the presentation model. These 3-D objects could be anchored in the real world in the grounding space. So even though the histograms are placed relatively in a 2-D coordinate system, they go beyond the presentation model in the third dimension. Observe that the ability to define new encodings (as we just did) without redefining the data model is a major benefit allowed by utilizing the encoding space. New encodings can be added or swapped in a modular fash-

ion by changing the encoding functions associated with a certain query operation.

In ARQuery a set of encoding functions and their corresponding parameters are generated for each query session operation. The results are rendered into the encoding space, and then grounded into reality in the grounding space. The encoding functions are described in more detail in the section on our visual querying grammar.

3.2 The Grounding Space

The encoding space provides static visual representations of the result set, but is in no way anchored in reality. The grounding space takes visual artifacts and anchors them in some dynamic coordinate system. In ARQuery, the query session table is tracked through the camera feed, meaning that the location of table components is changing as they move in the camera. Anchoring the visual encodings from the encoding space onto the moving table components is what defines the responsibility of the grounding space in our case. Note that because the visual encodings from the encoding space are produced according to a mapping to a presentation model, the grounding space must have an anchor for each component of the presentation model in order to define a complete mapping from visual artifacts to locations in the grounding space.

While in ARQuery we use visual tracking in order to anchor encodings in the camera feed, a more complicated mapping to anchors could be used. If a world tracking module was employed then encodings could be anchored on to the world space for persistence even when the table is not within the camera feed. The modularity of the grounding space definition allows for a wide range of such mappings.

4. INTERACTION MODEL

As described in our data model section, we model query sessions as a series of operations that successively modify the result set – namely selection, projection, groupby/aggregate, and ordering. Each operation has its parameters and an encoding function associated with it. Following an operation, the results are visually encoded and anchored in AR until another operation occurs, changing the result set and its encoding.

Gestural Query Specification: The query session operations are triggered by gestures on the encoded result set in AR. We adopt a gestural query specification that is inspired heavily on concepts from GestureDB [18], and defines the mapping from gestures to result set operations.

Mapping Gestures in AR to the Query Space: In order to map from gestures that occur in AR to operations in the query space we map from the grounding space to the query space. When creating visual encodings from operations on the underlying data, we map from the query space, to the encoding space, to the grounding space. When obtaining operations from gestures, we go in the opposite direction with the following process:

1. A gesture G occurs in AR that involves some sequence of key points K in the grounding space's target coordinate system.

2. K is mapped to a sequence P that corresponds to the presentation model components that occupy the points defined in K.

3. Using the inverse data model to presentation model mapping, P is mapped to query space operation parameters p_o for an operation that is determined using rules of the form $(G, p_o) \implies o$, where o is a query session operation. The antecedent (G, p_o) corresponds with a gesture in the gestural query specification.

This process allows us to effectively obtain query session operations from gestures in AR, and defines a model for interacting with structured data in AR using gestures.

5. A VISUAL QUERYING GRAMMAR FOR AUGMENTED REALITY

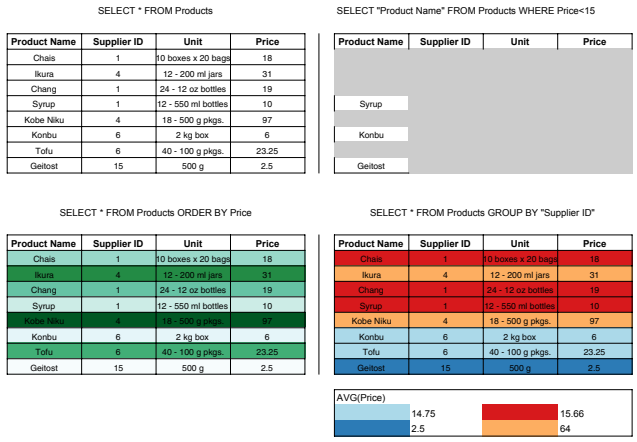

Figure 4: Query operations in ARQuery. Tapping and holding a value filters all rows that contain that value, excluding all others using a visual mask (selection). Tapping and holding a column header excludes the column (projection). Swiping up or down on a column header sorts the column, yielding a sequential color gradient. Swiping right on a column header performs an aggregation, yielding a divergent color annotation and a virtual result pane.

We construct an interactive visual querying grammar, building on previous work [18, 25]. From a query experience standpoint, a user is presented with an interface in which they view a table is present on ARQuery's video feed. Our system detects the table and extracts the query space data representation of the table schema, along with the corresponding presentation model. Following the extraction, the tracking of the table at the cell level commences and is maintained in the grounding space. Once the table is being tracked, the user can perform a gesture on the table in the video feed in order to carry out a query. The query is then executed in the query space, the result of which is stored as a materialized view. Using the query operation, the result set, and the presentation model, we then derive the *visual encoding* for the results using encoding functions. Once the visual encoding has been anchored in the grounding space, we render results in to reality (the live camera feed).

5.1 Visual Encoding for Real-world Data: Augmented Highlights and Virtual Results

As described earlier, mixed reality environments impose constraints that are not present in virtual environments, and require that result set encodings are additive. To do this, we devise the concept of two classes of additive visual encoding elements, *augmented highlights* and *virtual results*, as shown in Figure 1. These visual encodings maintain a strict connection to individual cells in the original table's presentation model in order to be tracked and rendered correctly in realtime.

Augmented highlights utilize visual cues (occlusion masks, colors, text annotations) to encode information. These are used above existing data elements in the real-world data table, in order to modify the meaning of these elements.

Virtual results on the other hand are "synthetic" visual artifacts that do not fit the shape or schema of the real-world data table. In Figure 5, the first four columns (orange/red), and the last 3 rows (masked grey) are augmented highlights, representing grouping and filtering respectively. The last column `(AVG(Price))` is a virtual result, which does not exist in the real-world table, and "floats" as a virtual element next to it. Note that virtual results can also be highlighted, as depicted with the green `ORDER BY` gradient coloring.

SELECT * FROM Products GROUP BY "Supplier ID" HAVING AVG(Price) > 15

Product Name	Supplier ID	Unit	Price	AVG(Price) ↓
Chais	1	10 boxes x 20 bags	18	15.6
Ikura	4	12 - 200 ml jars	31	64
Chang	1	24 - 12 oz bottles	19	15.6
Syrup	1	12 - 550 ml bottles	10	15.6
Kobe Niku	4	18 - 500 g pkgs.	97	64

Figure 5: Example of augmented highlights (first 4 columns, last 3 masked rows) and virtual results (last column). GROUP BY, HAVING, and ORDER BY queries are represented in our encoding using augmented highlights and virtual results.

5.2 Encoding Functions

The encoding functions produce the visual encoding elements in the coordinate space defined by a presentation model. Each of the functions is associated with a set of query session operations, and uses its associated operation along with the result set from the query space in order to produce the visual encoding. The encoding functions that we use in ARQuery are described below. While we found these functions to work well in our studies, it is important to note that a different set of encoding function may be used to satisfy differing requirements and produce another encoding.

exclude(rowIndex, columnIndex) - Excludes attributes, tuples, or individual cells from the table. This can be used to exclude tuples for filtering, exclude attributes for projection, or to exclude individual cells within a tuple to denote a null value. When excluding a full row, only the *rowIndex* provided. Excluding a full column is done similarly with *columnIndex*. Exclusion of a table component is represented visually with a white occlusion over the component.

group(rows) - Groups the specified *rows* within the relation defined by the query session table. Note that the group of a row is a one-to-one relationship, so a row cannot be grouped twice. Visually, grouping is done through the use of distinct color hues.

appendRow(tuple) - Appends *tuple* as a row to the table. The *tuple* parameter must fit into the schema of the table it is being appended to. Appended rows are visually encoded using synthetic rows that are drawn on the bottom edge of the table, and thus they are virtual results.

appendColumn(attributeName, attributeValues) - Appends the *attributeValues* sequence to the table in a column with a header defined by *attributeName*. The number of values in *attributeValues* must match the number of tuples in the table that is being appended to. Appended columns are encoded similarly to appended rows, also as virtual results.

associate(groupValues, valueType) - Associates a value with each group through the *groupValues* parameter, a mapping from each group to a value. The *valueType* parameter describes the values that are being associated with groups (`AVG(Value)` for example). Association is used in order to encode aggregate results. Legends are used in order to visually encode the values that are associated with a group, where each group is identified by its color and the associated value is displayed next to the color. The *valueType* descriptor is used as a legend header to give context to legend values. Figure 4 shows an example legend for associating `AVG(Price)` values.

appendAggregate(groupValues, valueType) - This is a special case of *appendColumn*, and is defined for convenience in representing aggregate results. This function Appends *groupValues* to the table in a column defined by the attribute *valueType*. For each row in the table, its value for the *valueType* attribute is determined by its group and its associated value in the *groupValues* mapping. By appending a column for aggregate results, you can perform the equivalent of a HAVING query by filtering on the appended column. Sorting by the aggregate results can also be done by utilizing an appended column, as demonstrated in Figure 5.

order(column) - Order is defined for three cases. In our definition we use the term synthetic to describe when the type of *column* is a virtual result (a column that was added to the end of the table). If the query session table is not grouped then all rows in the table are ordered globally. When the table is grouped and *column* does not represent an aggregate value, the rows in the table are ordered within their respective groups according to attribute defined by *column*. When *column* is synthetic and represents an aggregate value, the rows are ordered within the table globally according to the attribute defined by *column*. Note that synthetic rows are used to represent aggregate results for separate groups so by ordering globally we allow for ordering based on aggregate results. Visually, ordering is represented using the notion of color value from the HSV color model.

5.3 Encoding Joins

We now utilize the grammar above to define a visual encoding for join queries. In ARQuery, the table for a given query session could be joined with another relational table to produce a new result set. While not yet implemented, we define an encoding for joins for completeness.

In keeping with the additivity in AR constraint we defined previously, we construct all joins as left outer joins: $R_q \rtimes\!\bowtie R_j$, where R_q is the current query session's table and R_j is the table being joined with. This operation produces a resultset that contains all matching tuples without excluding tuples from the left operand of the join, which is the query session table in our case. Thus left outer joins are inherently additive on the query session table, and fit naturally into our additive visual encoding. Note that we also could encode other joins (inner joins, right outer joins, etc.) with the addition of extra exclusions and row and column appending, but the overuse of these operations clutters the context of the table in reality. For these reasons, we adopt left outer joins.

Consider the result of a left outer join $R_r = R_q \rtimes\!\bowtie_{a\theta b} R_j$, where $a \in \mathit{Attributes}(R_q)$, $b \in \mathit{Attributes}(R_j)$, and $a\,\theta\,b$ is the join predicate. For any tuple $t_q \in R_q$, define:

$$\text{matches}(t_q,\, x,\, y,\, \theta) = \{t_r \in R_r : x(t_q)\ \ \theta\ \ y(t_r)\} \quad (1)$$

which is the set of matches for a tuple t_q in the query session table using the join predicate $x\,\theta\,y$. Additionally, define:

$$\text{leftTuplesWithSameAttribute}(t_q,\, x) = \sigma_{x=x(t_q)}(R_q) \quad (2)$$

which is the set of tuples in the query session table with the same value for attribute x as the tuple t_q. In order to derive a visual encoding for joins, we consider three important cases for $\mathit{matches}(t_q, x, y, \theta)$.

For some join between R_q and R_j, consider a tuple $t_q \in R_q$ and a join predicate $x\,\theta\,y$. Let m be the result of equation 1 with parameters t_q, x, y, and theta. Then let l be the result of equation 2 with parameters t_q and x.

Case: $0 < |m| \leq |l|$. In this case, we can associate every tuple in l from the query session table with a tuple from the result of the join under the predicate $x\,\theta\,y$, with no leftovers. By using the *appendColumn* encoding function for each attribute gained in the join that was not originally in the query session table, we capture all the information in m. Observe that this case can be used to encode joins with the following data relationships: 1 to 1, many to 1.

Case: $|m| = 0$. In this case, there is not a matching tuple for t_q in the result of the join under the predicate $x\,\theta\,y$. Because we use left outer join, we must encode this lack of a match as a null value. In order to encode the null values that result from this join, we use the *exclude* encoding function on cells in the columns we appended for the join for each tuple that cannot be associated with a match. Using this case in combination with the first, we can encode joins with 1 to 0 or 1 relationships, and many to 0 or 1 relationships.

Case: $|m| > |l|$. In this case, we can associate every tuple in l from the query session table with a different tuple from the result of the join under the predicate $x\,\theta\,y$, but there are leftover matching tuples from the join result. We handle the tuples that can be associated with a matching tuple (the tuples that aren't leftover) the same way we did in the first case. In order to encode the additional matches, we use the *appendRow* encoding function with these tuples, which have attributes contained in $R_q \cup R_j$. Observe that this case enables the encoding of 1 to many relationships, and when combined with the second and first approach we can encode 1 to 0 or many relationships as well.

Considering these three cases for every tuple in a query session table, We define an encoding for left outer θ-joins. Additionally, vlookup queries can be encoded and we model them as a left outer join composed with a filter and a projection, producing one value. The value is added as a virtual result over top of the cell that the vlookup query was predicated on.

Join Specification: We consider the problem of join specification a separate problem, which is domain specific. For joining a query session table with a table that is within close proximity to it in the grounding space, The query model for GestureDB [18] could be adapted, allowing for gesture based join specification. If, however, a join of the query session table with a table in some back-end data store is desired, another form of specification would be required. This would require a method for specifying which table you would like to join with in the data store, and a method for specifying the predicate to join the tables with.

5.4 Going Beyond Tables

The implementation of ARQuery that we have built operates on real-world tables that inherently fit in to the relational model. While it may seem that this limits the use of ARQuery to a small subset of structured data, it should be noted that non-tabular layouts are well within the realm of possibilities for ARQuery. Extending ARQuery to other layouts is equivalent to changing presentation models in the visual querying grammar we have defined.

Consider extending ARQuery for use with restaurant menus. The menu has sections for different food types (entrees, appetizers, etc.), and each section has food items, along with prices; this is our presentation model. In order to augment this data according to relational operations, a mapping from presentation model components to data model components is required, which will allow us to represent the menu in our relational data space. The inverse of this mapping is used to produce the visual encodings of the result sets. With the addition of a query specification (in any modality, be it gestural, speech, etc.), we have everything we need in order to query the structured data that is the restaurant menu.

6. SYSTEM DESIGN

Figure 1 gives an overview of the ARQuery system, the individual components are described as follows. ARQuery taps into the device's camera video feed, and each frame is processed by our computer vision module to detect, segment, and filter tables. These tables are then OCR'ed and extracted into a typed 2-D array, which facilitates correct `ORDER BY` and `GROUP BY` operations. For extraction at the computer vision level, by detecting parallel lines, straight lines, and co-aligned text groups, we use a simple table detection method that makes the assumption that the table is the maximally connected component in the area determined to contain a table. This step is made performant by tracking objects across frames, minimizing the number of times the extraction pipeline is called.

Table Extraction: The transcribed 2-D array is then sent to the table extraction module, which is similar to efforts such as DeepDive [27] and Fonduer [26]. Due to the nature of our gestural query specification, we make the assumption that a header row exists, and the header row text and the contents of the table are then used to determine the schema of the table at hand. ARQuery further assumes a "row major"

visual layout, and uses this assumption to convert the typed 2-dimensional array provided into a SQL table.

Table Tracking and Rendering: A critical component of augmented reality is to *track* the rendered augmented layer to the camera feed in realtime. This is done by finding the contours of the table cells, and then subselecting all corner points of the table in an optical flow tracking using the Lucas-Kanade method. This keeps the grounding space representations up to date and allows the user to continuously move the camera view, but still have the visual encoding of the current query session be correctly grounded on to the real-world data.

Interactive Query Session: Due to the small size of the data in view, the interactive query session is maintained as a sequence of a configurable number of materialized views. Each operation is performed as a `CREATE MATERIALIZED VIEW` t_{i+1} `AS SELECT...`$Q_{i+1}(t_i)$. This allows users to use two fingers to swipe outside the table view, moving back and forth through the query session by performing a succession of "undo" and "redo" steps.

7. EXPERIMENTAL EVALUATION

Experimental Setup: In order to evaluate the *ARQuery* system, we compared it against two typical methods of analyzing real-world tabular data: using Microsoft Excel, and performing the analysis manually (i.e., by hand, without any digital tools). We chose Microsoft Excel since it is the most popular querying interface that supports *filter*, *sort*, and *group* operations (grouping performed using *pivot*), and is the typical tool to use in ad-hoc settings. For these operations, database frontends typically use interface paradigms similar to Excel (or directly connect to Excel), hence making Excel a good reference for evaluation. A *within-subjects* design was employed for this experiment, which is appropriate as the three methods that were being compared are distinguishable and carryover effects should be minimal. Our studies were conducted with 15 users recruited from the Columbus area with median age of 23, 8 of which were females and 7 were males, a sample size consistent with prior efforts in evaluating interactive data systems [16]. We used our iOS implementation of ARQuery for user studies, on an Apple iPad Pro 10.5″ device. We used a synthetic table of student data with seven columns and seven rows (plus an additional attributes header row) for testing. To stay consistent amongst users, each was given the same tutorial for each tool that was used in testing (Excel and ARQuery). In addition, each user was given a chance to get familiar using these tools before the study. We consider biases and confounding factors that are common when evaluating interactive data systems: learning, interference and fatigue [16]. By altering the order in which each system was evaluated for each user, we counterbalance carryover effects and reduce learning and interference effects. These effects were reduced further by providing a new table with shuffled rows for each query, and randomizing the order of the queries the users were asked to perform. To handle fatigue, the users were offered breaks between tasks.

For our tests, the user flow for ARQuery is as follows: the user holds an iPad in camera view in front of a printed sheet of paper with our dataset (Figure 1). A table is detected automatically and the table inference stage is run, deciphering the "shape" of the table. After this, the user performs their query session, issuing queries over the data by using the gestures described in the previous sections on the iPad touchscreen. The user ends their session once the query tasks are completed.

7.1 Performance

We were able to achieve perceptibly instantaneous [4] performance by providing immediate visual feedback well within 100ms. Upon testing over the user query sessions, we observed that ARQuery runs consistently at least 15fps (67ms per frame) while table tracking, which is a continuous operation for the lifetime of the query session. We found this performance to scale well for increased data size, we found the frame rates to stay over 15fps even when we doubled the cells to a 12×10 table. At the beginning of the session, the table inference stage (when ARQuery gains an understanding of the schema and data in the camera view) is run, which takes 68ms. During the same stage, a one-time OCR step (using the Tesseract library) took an average of 1.3s; putting the total initial setup time at under 2 seconds. The system-facing latencies for gesture detection and query specification, which includes the updating of the model that results from the specified queries, are negligible (under 1ms per frame).

7.2 Task Completion Time: Comparing with other methods

We measured the time it takes users to perform queries with three different methods on three different classes of queries. The three classes of queries tested were *Filter*, *Sort* and *Group By/Aggregate* queries. The users were given three tasks in the form of queries to perform using the three differing mediums. Queries were provided in natural language in both written and spoken form. Time was counted after the query has been read by the user and read out loud, and is stopped when the user indicates they have obtained the answer (i.e., says "Done").

This metric (completion time) serves as our objective measure of efficacy. In our experiment, we define completion time as the time it takes a user to go from a resting state to obtaining the answer to a query (for ARQuery, this includes the table inference and other system-facing times listed in the previous paragraph). For each class of query, users were read out loud and shown a query as an English sentence, and timed on how long it took for them to return the answer to the query. It should be noted that for Excel, the spreadsheets were already populated: we do not consider the spreadsheet input time into our completion times (Excel would be shown to be even slower if that were considered).

As we can see in Figure 6, when comparing ARQuery to the other two systems we found that ARQuery has a lower average completion time as well as a lower variance, denoting a faster and more consistent user experience, especially for group/aggregate queries. Furthermore, variances typically increase over the three query classes for the 3 different methods, but are quite small for ARQuery across the board. In order to show the statistical significance of the differences between ARQuery and other systems, we performed 1-tailed t-tests and confirmed that users perform the queries significantly faster with ARQuery than the with the other two methods with all significance levels (P-values) under 0.05.

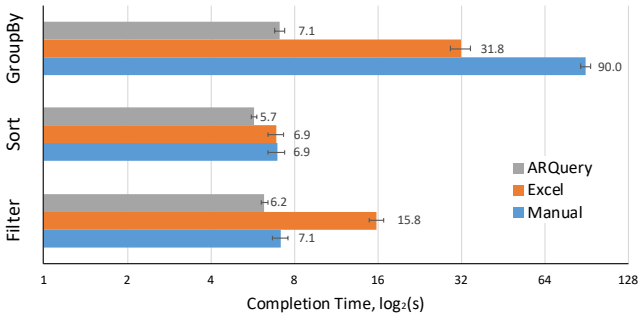

Figure 6: Average completion type for three classes of queries performed with different methods. Completion time for ARQuery was significantly faster than that of the other two methods.

7.3 Insights

ARQuery's average completion time for all queries tested (Filter, Sort, and Group/Aggregate queries) was lower (ie., a faster querying experience) and had less variance (i.e., a more consistent querying experience) when compared to other the other two query methods (with no tools a.k.a. manual, or with Excel). We observed there to be some variance in completion times across query classes, which correlates with the complexity of the interaction involved: sort queries were the fastest to perform, followed by filter, followed by group by/aggregate. This ordering is consistent with the other methods, with the exception of *filter* in manual (no tool), which reflects the human ability to quickly find things by browsing. While it seems at first that performing filter and sort queries by hand (manually) might be quicker than using ARQuery, our study results show otherwise. We expect this speedup to increase as the dataset gets larger and more complex – while ARQuery's times are expected to remain relatively constant or scale sub-linearly, the human cognitive costs [2] towards grappling with large and complex data are known to not scale well. From an ingestion standpoint prior to querying, it should be noted that while we included table inference times (less than 2s) in ARQuery's user completion times, time taken to populate the Excel spreadsheets was not factored in. Doing this would have found Excel to be even slower: we estimated (using the KLM model [5] and manual tests) that data input would have taken approximately 97 seconds for the data used in our study. Another subjective insight gathered from the user studies was that users were hesitant about their answers when performing manual calculations, and unsure if they had made mistakes. Thus, ARQuery (when used through a touchscreen, or through a headset) has the power to not only expedite user queries, but also to empower them with confidence and reduce possible human error. Overall, we find ARQuery to be performant for typical datasets, and it the fastest way to perform basic analytical queries over real-world data. Further, we observe that users feel more *confident* answering questions when ARQuery is enabled.

7.4 Future Evaluation

In the future we hope to test our hypothesis that task completion time for ARQuery grows sub-linearly with increasing data size. Additionally, we would like to consider common quantitative metrics such as usability and learnability. These qualitative measures could help to provide insight in to the perception of ARQuery and how it compares to similar systems that do not utilize augmented reality. Further, our observation of the increased confidence while using ARQuery warrants further investigation, bringing about the possibility of exploring a new metric for the evaluation of interactive data systems.

8. CONCLUSION AND FUTURE WORK

Performing ad-hoc analytical queries over *real-world data* – either in paper or digital form – is challenging. With the recent advances in modern computer vision, and with the pervasive availability of camera-enabled devices, we look towards using augmented reality as a popular way to query the real world. By translating query results into visual encodings, and then grounding them in reality, the user is able to "hallucinate" analytical query sessions and answer filter, sort, and aggregation queries faster and more effectively than performing the action by hand, or with a spreadsheet-based query interface.

While ARQuery is able to perform well with typical datasets and traditional queries, we foresee several exciting avenues of future work. ARQuery currently handles unary operations; performing truly binary operations such as joins can be added on by enabling ARQuery on multiple juxtaposed objects (e.g., holding a calorie table next to a restaurant menu), similar to the gesture-driven action in GestureDB [18], which has been shown to be easy-to-use for non-expert users. Further, ARQuery only supports the relational model – being able to add support for nested relational, hierarchical, and other data models are straightforward: this would entail creating a polygon-to-data inference mechanism and an interaction mapping for each data model. Finally, from a visualization standpoint, supporting 2-D and 3-D visualizations are a trivial next step (while histograms and pie charts are already supported in ARQuery, an interaction grammar or heuristic needs to be articulated about *which* visualization to generate for each column – visualization recommendation is itself an orthogonal and separate area of research). Finally, we are building support for a *spatial mode* into ARQuery, allowing users to map result visualizations onto real-world 2-dimensional and 3-dimensional spaces, allowing for immersive, context-specific analytics.

Acknowledgment: This material is based upon work supported by the National Science Foundation under Grant Nos. IIS-1527779 and CAREER IIS-1453582.

9. REFERENCES

[1] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent Advances in Augmented Reality. *IEEE CG&A*, 2001.

[2] J. Baker, D. Jones, and J. Burkman. Using Visual Representations of Data to Enhance Sensemaking in Data Exploration Tasks. *JAIS*, 2009.

[3] J. Bernardes, R. Tori, R. Nakamura, D. Calife, and A. Tomoyose. Augmented Reality Games. *Extending Experiences*, 2008.

[4] S. K. Card. *The Psychology of Human-Computer Interaction*. CRC Press, 2017.

[5] S. K. Card, T. P. Moran, and A. Newell. The Keystroke-level Model for User Performance Time

with Interactive Systems. *CACM*, 1980.

[6] C. Chabot, C. Stolte, and P. Hanrahan. Tableau Software. 2003.

[7] K. Chen, A. Kannan, Y. Yano, J. M. Hellerstein, and T. S. Parikh. Shreddr: Pipelined Paper Digitization for Low-resource Organizations. *ACM DEV*, 2012.

[8] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive Analytics through Pen and Touch. *VLDB*, 2015.

[9] D. W. Deaton and R. G. Gabriel. Check Reader Method and System for Reading Check MICR Code, 1993. US Patent 5,237,620.

[10] Q. Dong. Skip the Line: Restaurant Wait Times on Search and Maps. *Google Blog*, 2017.

[11] K.-S. Fu and J. Mui. A Survey on Image Segmentation. *Pattern Recognition*, 1981.

[12] H. Fuchs, M. A. Livingston, R. Raskar, K. Keller, J. R. Crawford, P. Rademacher, S. H. Drake, and A. A. Meyer. Augmented Reality Visualization for Laparoscopic Surgery. *MICCAI*, 1998.

[13] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. *SIGMOD*, 2006.

[14] A. Henrysson and M. Ollila. Umar: Ubiquitous Mobile Augmented Reality. *MUM*, 2004.

[15] S. Idreos and E. Liarou. dbTouch: Analytics at your Fingertips. *CIDR*, 2013.

[16] L. Jiang, P. Rahman, and A. Nandi. Evaluating Interactive Data Systems: Workloads, Metrics, and Guidelines. *SIGMOD*, 2018.

[17] M. Khan, L. Xu, A. Nandi, and J. M. Hellerstein. Data Tweening: Incremental Visualization of Data Transforms. *VLDB*, 2017.

[18] A. Nandi. Querying Without Keyboards. *CIDR*, 2013.

[19] A. Nandi, L. Jiang, and M. Mandel. Gestural Query Specification. *VLDB*, 2013.

[20] J. M. Rzeszotarski and A. Kittur. Kinetica: Naturalistic Multi-touch Data Visualization. *SIGCHI*, 2014.

[21] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *TVCG*, 2002.

[22] R. Szeliski. *Computer Vision: Algorithms and Applications*. 2010.

[23] G. R. Vesto. Augmented Reality Enhanced Triage Systems and Methods for Emergency Medical Services, 2011. US Patent App. 13/204,524.

[24] C. Wang, Y. Feng, Q. Guo, Z. Li, K. Liu, Z. Tang, A. K. Tung, L. Wu, and Y. Zheng. ARShop: a Cloud-based Augmented Reality System for Shopping. *VLDB*, 2017.

[25] H. Wickham. A Layered Grammar of Graphics. *Computational & Graphical Statistics*, 2010.

[26] S. Wu, L. Hsiao, X. Cheng, B. Hancock, T. Rekatsinas, P. Levis, and C. Ré. Fonduer: Knowledge Base Construction from Richly Formatted Data. *SIGMOD*, 2018.

[27] C. Zhang. DeepDive: a Data Management System for Automatic Knowledge Base Construction. *Madison, Wisconsin: University of Wisconsin-Madison*, 2015.

[28] E. Zhu, A. Hadadgar, I. Masiello, and N. Zary. Augmented Reality in Healthcare Education: An Integrative Review. *PeerJ*, 2014.