

## Introduction

There are many situations in which data is organized in tables. For example, a rural clinic might organize its doctors' schedules as tables of times and patient names. Organizations of fishermen or farmers might track market prices and the amounts of available goods in a pair of tables. Health officials planning a vaccine deployment in a developing country might track refrigerator inventories in a set of related tables (e.g., tables for refrigerator models, refrigerators, clinics and hospitals, and administrative districts). Although tables are a fine way to organize this information, spreadsheet applications are not necessarily the best way to manage and access these tables. The clinic's doctors would want to see a schedule of their appointments for each day. The fishermen and farmers could benefit from being able to query the organization's database through SMS messages to find the best market for their goods. The cold chain logisticians would need their tables to connect to each other and to see detailed information about individual items of data. These situations call for more than a set of spreadsheets, but the features they require can be generalized to common needs.

ODK Tables was built with two sets of goals in mind. Firstly, to produce a data management application that could be customized by the user with minimal overhead. To this end, ODK Tables provides a number of built-in views, and allows users to create their own views with HTML and JavaScript. These views can pull data from, and link to, other tables, so that users can form an integrated app, rather than a set of loosely connected (or completely disconnected) tables. ODK Tables also includes commonly-needed features (such as mapping) for convenience. Secondly, ODK Tables was designed for use in the developing world. For this reason, it was developed for Android devices, because mobile phones (being less expensive and more portable than laptops) are frequently used to collect data in the developing world. Also, ODK Tables includes an SMS interface, so that users with only a feature phone can add rows or query for data.

## Creating Tables and Adding Data

When first launching the app, the user starts at the table manager screen; from there, the user can create new tables, delete tables, and access the table properties manager. When creating a new table, users have several options. If a user has data to import (such as from an Excel spreadsheet), he can put a CSV file on the phone's SD card and use ODK Tables' CSV importer to create a new table with the columns and data from the CSV file. Users can export CSVs from ODK Tables with table settings included, and can then re-import it to a new phone to create a table with columns, data, and settings from the original table. Lastly, a user can create an empty table with no initial columns, and construct the table by adding each column individually from within the app.

When setting up columns, users can set some options on a per-column basis. Each column can have a data type (e.g., number or date range), which is used to restrict the values that can be put in the column. If the user wishes to restrict input to a limited number of specific strings (such as the multiple-choice questions found on many paper forms), he can choose the multiple-choice data type and create a list of options for the column. Users can also set up an abbreviation for the column (e.g., a column for dates of birth might have "dob" for an abbreviation), which can be used for convenience in searching and SMS messages.

### *Database Structure*

On its first launch, ODK Tables sets up a SQLite file on the phone's SD card, which holds most settings information and all user data. Initially, the database has only two tables: one for table properties (tableProps) and one for column properties (colProps). The tableProps table contains information about the user-created tables, and has columns for data such as the table ID, the table's display name, and view settings specific to that table. The colProps table holds information about columns of user-created tables, and has columns for data including the ID of the table the column is in, the column ID, the column display name, and the column data type. When a new table is created, a row is added to tableProps and a new table is created for the user's data. When a column is added to the user's table, a row is added to colProps, and a new column is added to the data table.

### *Adding Rows*

If the user has it installed on their phone, ODK Tables can make use of ODK Collect, a tool for collecting data with forms, to add and edit rows. If ODK Collect is installed, the row add button will create a form with fields corresponding to the columns in the table, launch ODK Collect, and fill in a new row with data from the Collect instance when it returns. Users can also set up forms with which to edit rows: if a column is designated as an ODK Collect column, values in that column are treated as form file names, and users can launch Collect with that form. On returning to ODK Tables, the values from the Collect instance fields that match table column names are filled in to that row. Alternatively, users can add and edit data directly from dialogs in ODK Tables.

### Browsing Data

#### *Collections*

Often, data needs to be organized into categories, such as grouping patient data by patient or grouping market prices by the type of goods. Categories are particularly useful for keeping track of historical data: for example, if an organization is collecting temperature data every hour for a number of refrigerators, it would make sense to group that data by refrigerator. The organization would likely want to be able to see a) the most recent data for all refrigerators and b) all data for a particular refrigerator. To meet this need, ODK Tables allows users to designate columns as “index columns,” meaning they are used to divide data into groups. If they have marked at least one index column, users can have two views of their data: an overview, which shows one item from each group; and a collection view, which shows all the items in one group. A user can also designate a “sort column,” which is used to determine which row out of each group is shown in the overview (e.g., refrigerator data could be sorted by a timestamp column, so that the most recent reading for each refrigerator is shown in the overview).

#### *Searches*

Collections might not be the only way a user wants to browse their data, so users are also able to search a table with column-value pairs. For example, a search of patient records for Bob Smithson would look like “name:Bob Smithson,” and a search for a tuberculosis patient named Bob Smithson would look like “name:Bob Smithson illness:tuberculosis.” For more complex situations (such as cold-chain monitoring), users might want to perform searches that use data from multiple tables. As an example, suppose there were tables for administrative districts,

clinics, and spare parts, and that each clinic was in a particular district (specified by a `district_id` column in the clinic table) and each spare part was in a particular clinic (specified by a `clinic_id` column in the spare parts table). A clinic with a broken condenser could find a clinic in their district with a spare condenser using join searches. Such a search would be done on the clinics table, and would be as follows:

```
“district_id:districtX join:spare_parts (type:condenser) id/clinic_id”
```

This query searches the spare parts table for rows where the type is condenser, then joins the result with the clinic table, where the clinics table id matches the `clinic_id` from the spare parts table. “`district_id:districtX`” restricts the results to clinics in `districtX`. The user's search results are the clinics in `districtX` that have a spare condenser.

### *Implementation of Database Queries*

User searches are primarily handled by two classes: the `DbTable` class and the `Query` class. `DbTable` is the class that handles all access to user data tables, including queries, additions, deletions, and updates. Earlier in development, the `DbTable` class also handled building the SQL statements for queries to user tables, including queries for overview tables (which take one row from each of multiple categories, with the criteria for categorization determined by the user) or searches. As these queries became more complex (such as with the addition of the capacity for searching with table joins), the SQL building functionality was moved into a new class: the `Query` class.

An instance of the `Query` class represents a single query to a particular user table, including a set of constraints (such as the value in a column being equal to a given value, or less than a particular value); a column to sort by and a sort order (ascending or descending); and a set of joins (with the table to join with and the query information for the join). A query can be built in two ways: either piece-by-piece, by constructing a “blank” `Query` instance for a table and adding constraints and joins individually; or by constructing a `Query` instance with the text of a user query, which the new `Query` object will parse. When a string is expected to be in the user search format (either text from the search bar or a search string given by a custom view and used to launch a table activity), the `Query` object parses it. Since SMS messages are in a different format, they are parsed by another class, which builds its own `Query` object. The `Query` class can provide SQL for all queries on user tables, including regular queries for all rows matching the query, overview queries, and “group” queries used to determine footer values (discussed below).

While the other queries are fairly straightforward, the SQL for the overview tables require several nesting joins. As discussed above, users can mark columns to be used to categorize items in tables into collections. The user can then view either a collection view (all the items in a particular collection, meaning they all have the same given values for each of the marked columns) or an overview (which includes one item from each collection, determined by whatever column the user has marked as the sort column). As an example, suppose there is a table `t`, with columns `a` (a column used for categorization), `b` (marked as the sort column), and `c` (neither a categorization nor the sort column). If the user searched for rows where `c`'s value was 12, the SQL would look something like this (note that the `id` column holds a unique row ID):

```
1. SELECT id, a, b, c FROM t JOIN (
2.   SELECT MAX(id) FROM
3.     (SELECT a, MAX(b) FROM t WHERE c = 12 GROUP BY a) x
```

```

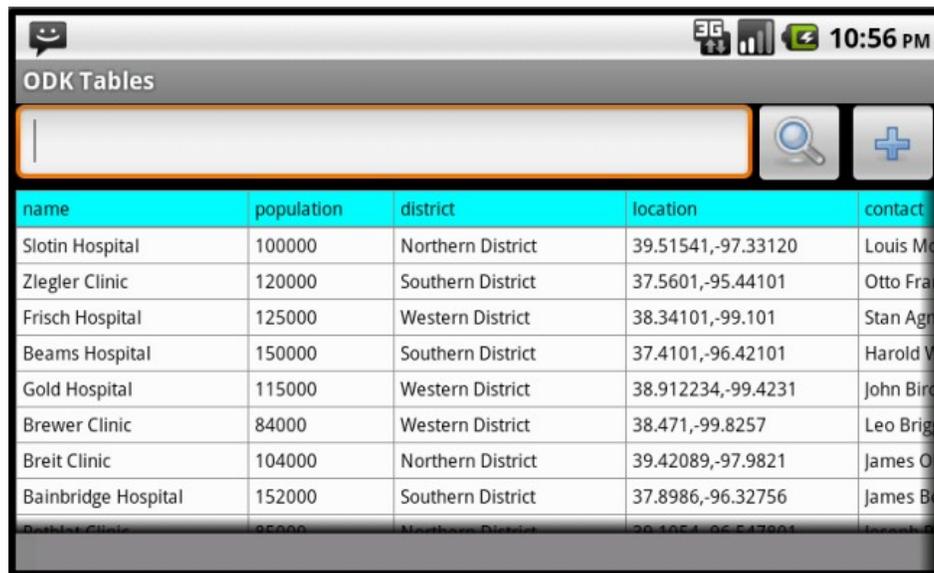
4.      JOIN
5.      (SELECT id, a, b, from t WHERE c = 12) y
6.      ON x.a = y.a AND x.b = y.b
7.      GROUP BY a
8.  ) z ON t.id = z.id

```

The first inner SELECT statement (on line 3) finds all values present in column a (which define the categories) and the maximum value in column b (the sort column) for each category. All rows in the eventual result will have one of these pairs of values for columns a and b. The SELECT on line 5 gets the IDs for all rows, then joins the two and takes the maximum ID from each category (this is to ensure that, if multiple rows in the same category have the same value in the sort column, no more than one row is taken from each category). Finally, these IDs are joined with the original table to get the data to be returned.

### Table View Options

ODK Tables provides a number of different views for user's data, and users can set which view they prefer on a per-table basis. If a user has set an index column, he can set options for overviews and collection views separately. The default view is the spreadsheet view, it being a natural way to view tables of data.



The screenshot shows the ODK Tables application interface. At the top, there is a search bar and a plus icon. Below the search bar is a table with the following data:

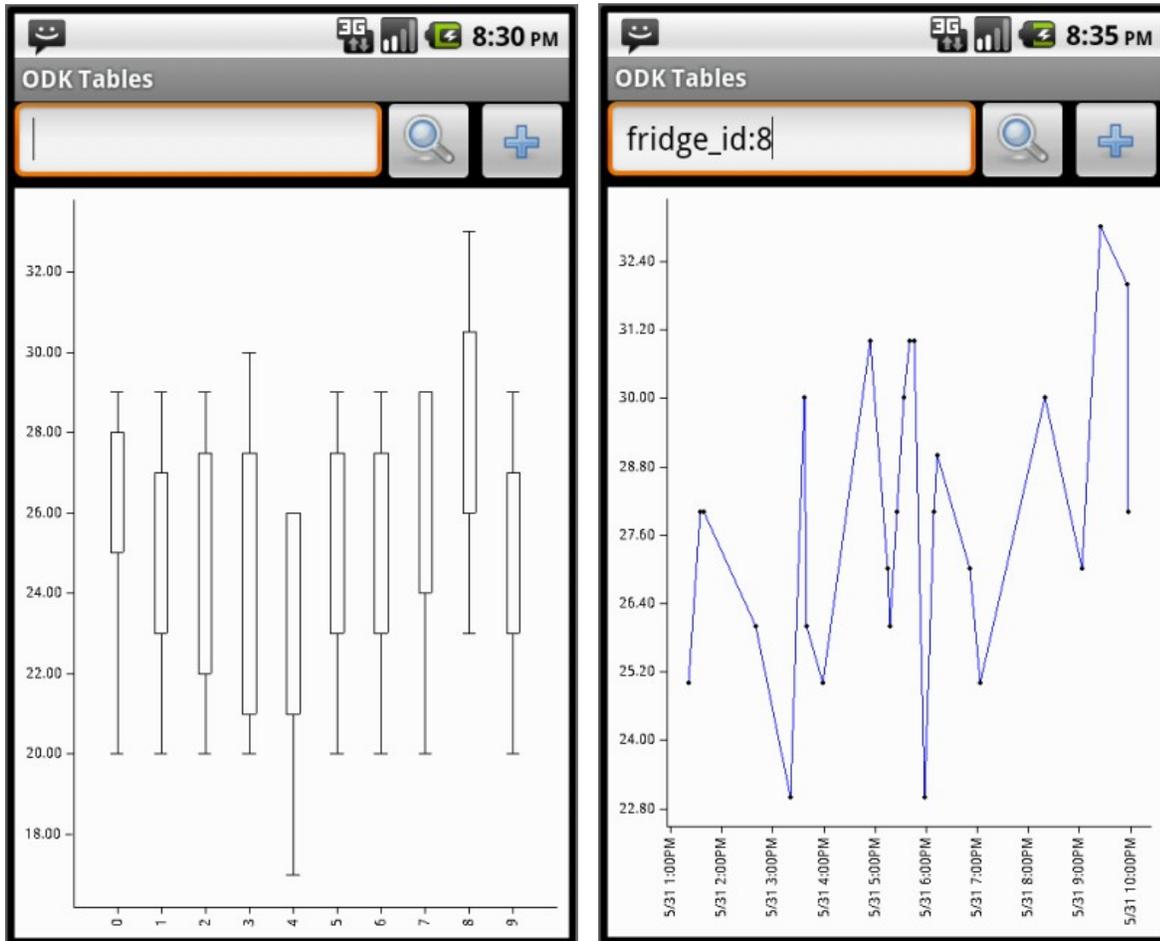
name	population	district	location	contact
Slotin Hospital	100000	Northern District	39.51541,-97.33120	Louis Mo
Zlegler Clinic	120000	Southern District	37.5601,-95.44101	Otto Fran
Frisch Hospital	125000	Western District	38.34101,-99.101	Stan Agr
Beams Hospital	150000	Southern District	37.4101,-96.42101	Harold V
Gold Hospital	115000	Western District	38.912234,-99.4231	John Birc
Brewer Clinic	84000	Western District	38.471,-99.8257	Leo Brigg
Breit Clinic	104000	Northern District	39.42089,-97.9821	James O
Bainbridge Hospital	152000	Southern District	37.8986,-96.32756	James Be
Dakota Clinic	85000	Northern District	39.1054,-95.547801	Leah B

Although ODK Tables does not include the ability to use complex functions to generate cell values, as a full spreadsheet application would, it does offer a rudimentary version of spreadsheet functions in the form of the footer mode. For each column, a user can choose a footer mode, which is the count, minimum, maximum, average, or sum of the values in that column; this value is displayed at the foot of the column.

Users can apply conditional coloring to the spreadsheets. Users can add rules to the spreadsheet, on a per column basis, with a value to which to compare the column's value, a means of comparison (such as equality or whether the column's value is less than the rule's value), and a color. For instance, in a table of refrigerator temperatures, a rule might be applied

to make the background of the temperature column red for rows where the temperature is greater than 40. Rules can be set for both cell background and cell font color.

ODK Tables has several graph views, including box-stem graphs and line graphs. The user designates columns to use for the x- and y-axes (when appropriate, the columns must be of the numeric type). For example, a user could use a box-stem graph for an overview of the temperatures of multiple refrigerators, and a line graph for a collection view of a single refrigerator's historical data.



ODK Tables also provides a calendar view and a map view. For the former, the user specifies a column of the date range type. The latter uses the Google Maps API for Android, and users specify a column of the location type. Location columns accept input as either latitudes and longitudes or in UTM format. As with the spreadsheet view, users can apply conditional coloring to the map view to determine the colors of the markers on the map.

### Custom Views

There are many situations in which users might want to create custom views for their tables. ODK Tables allows them to do this with knowledge of only HTML and JavaScript. Users must first prepare an HTML file on their computer, then transfer the file to their phone's SD card and specify the file name from ODK Tables' table properties screen. ODK Tables displays the

files in Android WebViews (a type of View built into Android that can render HTML and run JavaScript). Android allows Java objects to be made available as JavaScript objects in WebViews, and two such objects (called “data” and “control”) are available to the user's HTML. With these, the user's custom view can access data and open new views.

The user can create views both for individual items (rows) of data and for multiple items (tables). In the case of a view for an individual item, the “data” object can be used to access the value for each column in the row being viewed. In the case of a view for a table, the data object allows access to data by row number and column name. In both cases, the “control” object is also available, and can be used to interact with other parts of ODK Tables. The control object can be used to query for additional information (from any table), with the same user search format used in the search box. A table object (like the one automatically available for custom table views) is returned for use by the user's JavaScript. The control object can also be used to launch a new table activity (optionally, search text can be specified for the new table activity to use).

As an example, if an app for managing refrigerators in various hospitals had a table for hospitals and a table for refrigerators, users might want the custom item view for the hospital table to access the refrigerator table, so the user can see the refrigerators at that hospital. The control object could be used either to get the list of refrigerators (to be displayed directly in the hospital item view) or to allow a button to launch the refrigerator table with a search for refrigerators in the given hospital.



### *Spreadsheet View Implementation*

Originally, the spreadsheet view was implemented using Android `TableLayout`, which is a View class built into Android for displaying tables. Using a `TableLayout` involved creating an Android View object for each cell in the table to add to the `TableLayout`. This simplified some aspects of the code, since Android handled most of the work of laying out the table. Also, a unique ID could be assigned to each view object to make it easy for click listeners to determine which cell was clicked. However, using the `TableLayout` was slow for large tables, because a view object had to be constructed for each cell, causing the display time to grow with the size of the table. To avoid having to construct so many objects, the implementation was changed to override the View class's `onDraw()` method and draw the table directly on the Canvas. This made the display time significantly more reasonable, albeit at the expense of adding code that might be more inconvenient to maintain.

### SMS Interface

Because many people in the developing world have access to phones, but not to computers, organizations might want to use SMS to collect or distribute data. For example, a clinic could permit patients to schedule appointments through SMS, or a monitor connected to an inexpensive feature phone could be set up to send in readings (such as refrigerator temperatures). SMS has been the basis of numerous data collection tools for the developing world, including `FrontlineSMS`[3] and `RapidSMS` (which, among numerous other projects, is used by `ChildCount+` to monitor childhood illnesses with SMS messages[2]). SMS has also been used for querying, such as with `SMSFind`[1]. These projects are generally built for large-scale deployments and require additional equipment (such as laptops or servers). ODK Tables' SMS interface is designed for smaller-scale use: it requires only the phone, albeit at the cost of being limited by the phone's capacity to process incoming messages.

### *SMS Syntax*

By default, there are two formats for SMS messages: one for adding rows, and one for querying. For both, messages start with “@table\_name” to specify what table the message is meant for. For row additions, the table specifier is followed by a series of strings in the format “+column\_name value” to specify the values for each column in the new row. For instance, for a table of refrigerator temperature data, “@temps +fridge\_id 12 +temperature 17.3” would add a row for refrigerator 12 with a temperature of 17.3. For queries the table specifier is followed by a series of “?column\_name” strings (to indicate what column values should be included in the response) and strings in the format “(comparator)column\_name comparison\_value”. For example, “@temps ?fridge\_id >temperature 40” would return the refrigerator IDs of refrigerators with temperature readings above 40.

### *Shortcuts*

Because the syntax is difficult to remember and easy to mistype, ODK Tables allows users to specify their own formats for additions and queries by creating a table of shortcuts, which might be as follows:

name	input	output
record	fridge %id% is %temp% degrees	@temps +fridge_id %id% +temperature %temp%

request	temp of fridge %id%	@temps ?temperature =fridge_id %id%
---------	---------------------	-------------------------------------

The names in shortcut tables are recognized as possible values for the table specifier required for all messages. If the specifier for a message is the name of a shortcut, ODK Tables will attempt to match the message to the input format for that shortcut. Strings within percent signs are placeholders for the user's input, and the values that take their place are put into the output in place of the corresponding placeholders in the output format. For instance, if ODK Tables had been set with the shortcut table shown above, then a possible message would be “@record fridge 12 is 17.3 degrees”. Matching that against the input format for the “record” shortcut, Tables would determine 12 to be the value for the “id” placeholder and 17.3 to be the value for the “temp” placeholder, and then use the output format to produce a message using the default SMS syntax (“@temps +fridge\_id 12 +temperature 17.3”). ODK Tables can then parse the new message to determine what action to take.

### Performance

The main factor limiting the size of the data sets that ODK Tables can be used to manage is the time it takes to display a table. I ran performance tests for spreadsheet views, box-stem graphs, line graphs, and custom views. Performance tests were not run on map views (because their load time would depend significantly on the speed of the network connection used to load map tiles) or calendar views (since a calendar view with even a hundred items would be very difficult for the user to understand anyway).

The data displayed was a table like one that might be used to track refrigerator temperatures (with a column for a refrigerator ID number, a column for the temperature, and a column for the time). The data was generated somewhat randomly, with refrigerator IDs from 0 and 9 and temperatures between 20 and 30 (recorded to one decimal place), except that it was guaranteed that there would be an equal number of rows for each refrigerator ID, and that the times would be increasing.

#### *Default View, Searching, and Overviews*

Tests on the default view were done on a Motorola Droid running Android 2.2.2. Times reported are the mean of five trials.

The first set of performance tests were run with the default settings (no categorization was used, so the entire data set was displayed). The times are noticeable, but probably not so much as to make the application inconvenient to use.

Number of Rows	Average Time (ms)
10	1920.6
100	1983.2
1000	2272.6
2000	2444.4
5000	4738.2

The second set of performance tests timed searching. After the table was displayed, a

query for rows with “0” for the fridge ID was put in the search field, and the test timed how long it took to show the search results.

Number of Rows	Number of Results	Average Time (ms)
10	1	258.4
100	10	250.0
1000	100	332.4
2000	200	361.0
5000	500	304.2

The third set of performance tests were run for overviews, with the fridge\_id column used for categorization and the time column used for sorting (so that the overview showed the most recent temperature from each of the 10 refrigerators).

Number of Rows	Average Time (ms)
10	1874.8
100	1912.4
1000	2099.8
2000	2351.0
5000	2885.0

### *Graph Views*

Graph view tests were also run on a Motorola Droid running Android 2.2.2, and, again, times reported are the mean of five trials.

The box-stem graphs were tested with box-stem graphs set as the view type for overviews, so that a box and stems were shown for each of the 10 refrigerators, with fridge\_id on the x-axis and temperature on the y-axis.

Number of Rows	Rows per Box	Average Time (ms)
100	10	1928.4
1000	100	2066.4
2000	200	2517.6
5000	500	3700.6

The line graphs were tested for collection views (so that the rows for one refrigerator, a tenth of the rows in the database, were displayed as a line graph), with time on the x-axis and temperature on the y-axis.

Number of Rows	Rows per Graph	Average Time (ms)
100	10	486.2
1000	100	636.2
2000	200	899.0
5000	500	1065.0

### *Custom Views*

Due to the Droid being unavailable, performance tests on the custom views were run on an LG Optimus Slider (LG-VM701), which was a somewhat faster phone than the Droid, running Android 2.3.4. Because of difficulty instrumenting performance tests involving JavaScript in WebViews, the times reported are ranges of the approximate times for five trials.

Two different HTML files were tested. View 1 displayed a list of temperatures: for each row, it got the temperature and added a list item element to the list. View 2 displayed a table of temperatures: for each row, it got the fridge ID, temperature, and time, created table data elements for each, added them to a table row element, and added the table row element to the table. Both views were used to display the entire table.

Number of Rows	View 1	View 2
1000	1.8-2.1 seconds	3-3.5 seconds
5000	7-10 seconds	13-15 seconds

### Remaining Problems

#### *Rigid SMS Syntax*

User ability to conform to a very specific syntax for SMS messages is likely to be an obstacle to using SMS for querying and adding rows. For expectations of the success of rigid formats for short messages like these, we might look to a study of a Twitter-based disaster relief tool by Kate Starbird and Leysia Palen. Despite regular tweets about the syntax, a video explaining it, and a web-based tool for building tweets that conformed to the syntax, the tool saw little use from the on-the-ground users for whom it was intended[5]. Although allowing users to specify their own formats, which can be specific to the situation and perhaps closer to something that would seem natural, SMS messages for ODK Tables still must match those formats exactly. It would be beneficial for the SMS parser to be able to tolerate at least some variation, such as a typo in a column name. ODK Tables should also provide more helpful responses in the event of an invalid message.

#### *Single-Operation SMS Messages*

The SMS interface permits only one operation (a row addition or query) per message. Although many users in the developed world might have unlimited text messaging plans, this is not the case in the developing world. For instance, the makers of Uju (an SMS-based system that compresses messages to reduce cost) calculated that a theoretical system in Malawi that required one message per month for each HIV patient in the country would cost \$USD135,000 per

month[4]. It would be useful to permit multiple operations in a single message (either to reduce the cost to an organization using ODK Tables internally, or to encourage use of an ODK Tables-based service for people outside an organization).

### *Join Search Syntax*

The syntax for using joins in search is difficult to remember. Users specify the table to join with, a set of one or more pairs of column names (one from the table being searched, and one from the table to join with), and, optionally, a query to limit the results from the table being joined with. When the search syntax is being used by a custom view (which only has to be set up once), the difficulty of the syntax is less of a problem. However, there might be some cases when a user wants to use joins in their own searches. Since, in many cases, the columns that should be matched are likely to have the same names, it might be helpful for ODK Tables to assume it should match columns with the same name unless another set of column pairs is given (similar to how SQL databases match columns with the same name if nothing else is specified for a JOIN statement).

### *Language*

ODK Tables currently assumes the use of English, not only in the user interface, but also when it comes to user data (for instance, “now” is recognized as input for date columns). Additionally, the application uses American standards for date and time formatting. ODK Tables code currently has little in place to accommodate different language settings.

### *Possibility of Inconsistencies with Custom Views*

Because the custom HTML views and table and column settings are edited separately, it is possible for the two to become inconsistent. For example, if a custom item view accesses the value in a column called “refrigerator\_id”, and the user changes that column's display name to “fridge\_id”, the custom view will stop working. Additionally, this makes it more difficult than necessary for users to debug custom views that they are building, since the views currently fail silently if the user's JavaScript passes invalid arguments (such as a string expected to be a column name that does not identify any column) to a method.

### Works Cited

1. Chen, Jay, Lakshmi Subramanian, and Eric Brewer. 2010. “SMS-Based Web Search for Low-end Mobile Devices.” Paper presented at MobiCom, Chicago, IL, United States, September 20-24.
2. ChildCount. 2012. “ChildCount.org Home.” Accessed June 6. <http://www.childcount.org/>.
3. FrontlineSMS. 2012. “The Software.” Accessed June 6. <http://www.frontlinesms.com/the-software/>.
4. Lu, Wei-Chih, Matt Tierney, Jay Chen, Faiz Kazi, Alfredo Hubbard, Jesus Garcia Pasquel, Lakshminarayanan Subramanian, and Bharat Rao. 2010. “Uju: SMS-Based Applications Made Easy.” Paper presented at ACM Dev, London, United Kingdom, December 17-18.
5. Starbird, Kate and Leysia Palen. 2011. “Voluntweeters': Self-Organizing by Digital Volunteers in Times of Crisis.” Paper presented at CHI, Vancouver, BC, Canada, May 7-12.