# ODK Tables: Building Easily Customizable Information Applications on Android Devices

Waylon Brunette, Samuel Sudar, Nicholas Worden, Dylan Price,
Richard Anderson, Gaetano Borriello
Department of Computer Science and Engineering
University of Washington, Seattle, WA [USA]
{wrb, sudars, dylan, anderson, gaetano}@cse.uw.edu, nick.c.worden@gmail.com

## ABSTRACT

ODK Tables is an Android app that allows users to enter and curate tabular data. Users can explore the data through a variety of built-in views or build custom views using HTML/JavaScript. It also supports the linking of multiple data tables. Data values can be updated in a variety of ways, including using mobile data collection tools such as ODK Collect, that support rich data types including multi-media, or by communicating with low-cost phones over SMS. Additionally, ODK Tables supports a simple synchronization scheme appropriate for a distributed workforce and backed up on cloud servers. The goal of ODK Tables is to lower barriers to developing customized information applications by making it easy to customize data views using standard web technologies that do not require recompilation. Our experience in working with many organizations in the developing world led us to make feature choices based on their input (through an on-line survey) with particular consideration to the potential pool of developers available. In this paper, we report on our implementation of ODK Tables and some of its performance parameters. We have designed it to be a flexible solution for a variety of use cases, including logistics management, public health, and environment monitoring where previously collected data is often revisited and updated.

## Keywords

Open Data Kit, mobile phones, mobile database, spreadsheets, SMS, data tables, remote synchronization.

## 1. INTRODUCTION

Tools that simplify mobile data collection, such as Open Data Kit (ODK) [13, 26], have helped make smartphones a suitable platform for ICTD applications. With increasingly powerful mobile tools available, organizations desire more sophisticated data collection capabilities that go beyond simply replacing paper forms with a convenient user interface on a smartphone. For example, a rural clinic might organize schedules for doctors and nurses as a table with columns for time slots 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 linked tables (e.g., refrigerator inventories at clinics, clinics within a district, and parameters of refrigerator models).

ODK Tables [14] is intended as a general-purpose data management app for Android devices that can be customized by an organization, enabling them to deploy a range of information services. It occupies a space between data represented in spreadsheets (e.g., Excel) and relational databases (e.g., Access), the two most common general-purpose data management methods now in use in the developing world. However, we only include features that are commonly used across many domains and are relatively easy to explain to users. We exclude features that are highly specialized, error-prone, or just too cumbersome for the smaller screen size of a phone. Many advanced features can be implemented through extensions that organizations can deploy through standard HTML/Javascript/CSS. Thus, ODK Tables is a flexible, open source platform that enables workers in the field the ability to collect, manipulate, and view data on their Android devices, share data between devices, and save the data to the cloud.

ODK Tables enables users to aggregate, curate, analyze, and visualize their data on mobile devices. It provides a number of built-in views (e.g., table view, list view, map view, graph view) that can pull data from, and link to, other tables, so that users can form an integrated mobile application, rather than a set of loosely connected (or completely disconnected) tables. Data values can either be updated in Tables or can leverage ODK Collect, a widely used mobile data collection tool supporting rich data sets including location coordinates and multimedia.

Many information systems often require an internet connection to access data, which is challenging in developing regions where connectivity is not as reliable. Because ODK Tables caches data on the mobile device itself, it enables workers to explore and update relational data while in the field – even while disconnected. These intermittently-connected mobile devices can synchronize with a cloud-based server when possible and thus share portions of larger data tables.

ODK Tables includes an SMS interface, enabling users with a feature phone or no internet data plan to add rows or query for data. The smartphone caching a data table acts as a local hub with additional local expertise, namely, the field workers themselves, to further refine or structure the data before it is disseminated more widely through the server synchronization process. Thus, this synchronization scheme keeps data tables synchronized across a distributed workforce as well as the backup cloud servers.

This paper presents the ODK Tables platform for building mobile information applications for collecting, querying, updating, and reporting data. First, we discuss several target use cases that drove some of the requirements for ODK Tables. We then discuss the implementation and design decisions that help enable a varied set of applications, such as fine-grain access control and the user filtering mechanism for distributing large data tables to users, the client synchronization mechanism, and the conflict resolution strategy. Finally, we present a preliminary evaluation including basic performance numbers and discuss future project directions.

## 2. CASE STUDIES

There are many situations where data needs to be organized and shared between many users. A simple table is a common method of organizing data to be shared, as evidenced by the prevalence of simple Excel spreadsheets. While tables may be a good way to organize information, spreadsheet applications are not necessarily the best way to manage and access these tables. Doctors at a clinic might want to see a schedule of their appointments for each day. Fishermen and farmers could benefit from being able to query a database through SMS messages to find the best market for their goods. Cold chain logisticians need their tables to connect to each other so that they can judge if they will have adequate refrigeration capacity for a new vaccine campaign. These situations call for more than a set of simple spreadsheets, but not the extra complexity of a complete relational database system and its cumbersome SQL query language. To be of use to non-technical users, the platform needs to be intuitive and provide users with several ways of viewing data including entirely custom views developed by the users' organization. To determine a general set of requirements and the common features needed for ODK Tables, we studied five representative use cases with a particular focus on cold-chain monitoring.

### 2.1 Cold-Chain Monitoring

National immunization programs are an effective public health intervention that is essential for controlling diseases. An important component of these programs is the vaccine cold chain—the equipment (cold rooms, refrigerators, freezers, cold boxes) used to store the vaccines. It is essential to have adequate cold storage space to ensure that vaccines are kept in an appropriate temperature range from arrival in the country until they are delivered to patients. A cold chain inventory is a management tool for tracking the cold storage equipment used in the country's immunization system. A cold chain inventory consists of information about the health facilities (location, type of facility, role of facility in the immunization system, the size of the population served by the facility, and infrastructure information such as the availability of electricity and other fuels), as well as a list of the cold chain equipment for each facility. The key information about equipment comprises the model, fuel type, age, and working status. In some cases, the cold chain inventory is also used to track performance data such as the number of times the temperature is recorded as out of an acceptable range. Accurate knowledge of the vaccine cold chain allows identification of sites with inadequate storage, improved allocation of new resources, and makes it possible to plan for introduction of new vaccines.

Unfortunately, cold chain inventories are frequently out of date, with a large amount of inaccurate information. To address this, a system-wide update is needed, which includes facility visits where the assets are determined (possibly with respect to a reference list of equipment from an earlier inventory). The inventories are conducted by trained teams of enumerators who visit all facilities in a region. For a paper-based inventory, forms are filled out at each facility, collected, and sent to a central location for data entry. Eventually, all information is combined into the national database. The inventory information is often stored in spreadsheets. The Cold Chain Equipment Manager (CCEM) software [1] was developed to provide an improved management tool for Ministries of Health in working with the vaccine cold chain. The usefulness of CCEM is dependent on the extent to which the inventory can be kept up to date. A software system that supports a remote workforce making real-time updates on location will significantly improve the inventory update process. When field workers verify the cold chain, it would be helpful if instead of outdated paperwork they could use a mobile device that automatically downloads the subset of cold chain information for the district showing the most up to date information. Automatic updates may also decrease duplication of work between fieldworkers because a synchronized mobile device would always have the most current information.

### 2.2 Basic Search

A very common application is to help people find a specific piece of information in a larger index. For example, a table populated with weather forecasts could be queried based on location or date (or both), or a table of bus schedules could be queried by route and time. Applications need to be able to query with conditions such as "time > now" or ask for a specific piece of information by inputting an identifier (e.g., a bus stop number). An example ICTD project of this type is the *bus project [2] where users query transport arrival times via SMS.

### 2.3 Market Prices

One particular instance where both basic search and basic data collection could be useful is the exchange of price information [24]. For example, farmers or business agents can send information about their available crops to a phone storing values in a simple table, allowing buyers to query the table to find products they are seeking. Data rows might include type of crop, quantity, price, and location, permitting buyers to limit results to their area or search based on asking price.

### 2.4 Scheduling

Doctors and other skilled workers are often in heavy demand in the developing world. Managing and scheduling both personnel and shared resources such as equipment is necessary to ensure that these limited resources are not under-utilized. Therefore, the platform must be designed to make it easy to create a scheduling system. For example, if a work day is defined as 8am to 6pm and rows already exist with time range values from 9-10 and from 3-4, a response to a query for availability would include the time from 8-9, 10-3, and 4-6. These appointment options could be useful for a variety of small businesses, as well as other organizations like health clinics, to allow clients to query, set, and confirm their appointments.

### 2.5 Automated Data Collection

In addition to updating cold chain inventory with site visits, automated data about a refrigerator's current temperature can be collected and viewed on an Android device. To obtain automated information, organizations can deploy a sensor reporting tool such as FoneAstra [5] that can be used to attach sensors to low-cost mobile phones so that their sensor data can be reported via SMS. The data acquired (refrigerator's ID, temperature, time) can be sent automatically to a central server and/or a supervisor's smartphone. The data can be tracked, summarized, and graphed, allowing supervisors to check the status of all storage units from their own phones, as well as store and view historical information.

## 3. REQUIREMENTS

The requirements of this project evolved from a set of use cases for ICTD projects in the developing world (such as the subset described in the previous section) deployed by organizations that already employ existing ODK tools. Based on feedback from these users and their feature requests, as well as our own deployment experiences, we identified a common model for these use cases – namely, an organization with many field workers who need to share a single dataset consisting of multiple tables. The workers may need to view and update this dataset in the field, and the managers need to be able to distribute portions of the dataset and restrict access so that field workers only see and edit the parts that are relevant to them. Additionally, we conducted an online survey of ODK users where 85 respondents from around the world provided information on what were the most important features that would improve their experience with ODK, as well as what missing functionality prevented their adoption of the current ODK toolset for some of their projects. When asked to rate how important it was to "Keep data on a mobile device synchronized with a server" 55% of the respondents gave a rating of 5 corresponding to "Very Important", with a mean of 4 and standard deviation of 1.24.

Informed by the answers to questions such as these, we derived the following requirements that were not being adequately met:

- ***Add, delete, search, and update existing data, as well as custom queries to extract useful information.*** The ability to interact with the data, rather than merely collect it, was one of the most desired features requested by respondents to the online survey. Moreover, data tables can be linked (or joined, in database parlance) to connect two columns in two different tables. The ability to import large amounts of data to build a new table from a standard spreadsheet file (e.g., csv) and export data to the same file formats for easy interchange with other tools was also cited. Finally, the inability to pre-load an editable dataset, possibly using the structured ODK Collect form rendering tool, was the most common reason given for an organization choosing not to adopt ODK.

- ***Share data across devices and keep the ensemble synchronized.*** Besides simply backing up data, it should be possible for multiple devices to access and modify the same set of data (with subsets keyed to different clients), and the data should stay in sync as tightly as connectivity limitations permit. Conflict resolution will most likely require human judgment, and a good user interface is essential for accomplishing this task.

- ***View data in multiple ways.*** Built-in views to ODK Tables include spreadsheet, list, map, and graph views, but users always want more customization for a particular emphasis or a different subset of information. It is important that these custom views be possible without having to recompile the app. On the back end, the ability to aggregate and summarize data collected from multiple field workers was rated as "Very Important" by 60% of respondents.

- ***Robustness to sporadic network connectivity.*** The application should never rely on connectivity being available and must be able to cache data from the server as well as new data or edits entered by the user until connectivity is established. The synchronization protocol must deal gracefully with limited or no connectivity.

- ***Easy to deploy and maintain yet flexible enough to be customizable to different needs.*** Many organizations in the developing world do not have the ability to take on sophisticated IT responsibilities. The basic solution should be simple for less technical users to set up and keep running. However, it must be possible to customize as a project and the workforce matures using standard web technologies. Training costs must be kept low and be incremental as more advanced features are exploited or developed.

- ***Provide moderately fine-grained access control that allows both table and row-level permissions.*** Administrators should be able to restrict access to tables and metadata as well as easily subset tables so that each subset can be made accessible for a particular user or group of users. As most ODK users are already used to the data form model, their finest-grain size is a row of a table resprenting one instance of a form. The forms used for editing the data include finer-grain controls for restricting access to individual data fields. There must also be a simple mechanism for adminstrators to create groups of users and set access controls for each group.

- ***Maintain a complete history of changes to a table on the server.*** When rows are created, updated, or deleted, the history of these changes should be saved and made accessible to administrators for post-analysis of any problems that may arise.

- ***Scalability in terms of data and number of users.*** It must be possible to efficiently handle (UI interactions, file import/export, synchronization, etc.) moderately sized datasets (1000s of rows in tables with 10s of columns). Additionaly, deployments involving 100s of workers and client devices must not pose special challenges to the server infrastructure.

- ***Extensibility to other server backends.*** In the modular spirit of ODK, in general, it should be possible to adapt the server storage and synchronization protocol to different backends using a RESTful HTTP API. Similary, new clients (e.g., an embedded monitoring sensor) should be able to implement a small set of interactions to a server using the same API.

## 4. RELATED WORK

There are several smartphones apps that implement traditional spreadsheets (i.e., Excel-like functionality where formulas can be attached to individual cells). The name "ODK Tables" was specifically chosen to highlight the distinction from spreadsheets and emphasize the similarity to relational database tables. Formulas are all column-based rather than cell-based as in traditional spreadsheets making the presence of simple built-in formulas (e.g., max, mean, etc.) readily visible rather than hidden within a cell. Similarly, conditional formatting is also applied at the column level to ensure consistency.

Many projects support client-server replication systems for mobile computing such as Bayou [23], Coda [15], and PRACTI [4] that enable disconnected operation, but are not necessarily designed to manage data access restrictions for large numbers of users who need to share subsets of data. Commercial technologies such as Dropbox [10] have simplified synchronizing files between devices. These are not adequate for our users, who need a more fine-grained system, as small portions of data (just one row of a table) may be changing as field workers go about their daily tasks. Mobius [8] creates a mobile unifying and data serving abstraction for mobile apps that provides automated caching, predicates, notifications, and protocols for disconnected operation. Interestingly, Mobius presents the programming abstraction of a logical table of data that spans devices and clouds. However, Mobius is too heavy-weight a solution for our requirements as we do not need predicates or notifications because of our need to minimize the amount of network traffic to keep operating costs low and only synchronize when possible or requested by the user.

Additionally, other database applications have similar synchronization functionality; however, the currently available database solutions have drawbacks that make them a much less than ideal choice. For example, CouchDB [9] is a "database that completely embraces the web." As such, it has built-in access control and synchronization features and is accessible through an HTTP API. However, CouchDB is a JSON document-based database and does not cleanly map to the relational model of data in ODK Tables. The document paradigm means that each database is simply a collection of documents that are independent and schema-free. In ODK Tables, we wanted to retain the simple table concept of spreadsheets with typed columns; it is hard to map this to a representation in CouchDB in an efficient way. Since the relational model is so well understood and supported by Android's SQLite, it makes little sense to take on additional and unnecessary complexity.

Google Fusion Tables [12] is a service offered as part of Google Docs that provides access to simple cloud-based tables and presents an HTTP API based on SQL. The big disadvantage of Fusion Tables for the synchronization protocol is the inability to insert processing logic into the data path between a mobile device running ODK Tables and the backend database. This limits functionality to only the features provided by the Fusion Tables API. For instance, Fusion Tables does not provide any mechanisms for controlling concurrent access to the table, does not expose its access control model through an API, and does not provide permissions on a finer level than a whole table.

Tablecast [22] is an XML protocol built on top of the Atom syndication format [3]. It is designed specifically for dealing with concurrent modifications to tables of data. However, because it is based on the Atom specification, Tablecast relies on a publish/subscribe architecture where each participant acts as both a client and a server. Modifications to a table are published as a feed, and multiple feeds from multiple publishers can be merged together to see the latest state of a table. This is a bad architecture for the requirements of the synchronization protocol because many mobile devices (such as smartphones), with their limited battery life and network connectivity, can be unreliable as providers of feeds. Furthermore, the architecture would require phones to talk directly with each other, which is error-prone and cumbersome for the same reasons.

Oracle's Database Mobile Server [18] fulfills almost all of the stated requirements. It uses a relational model to synchronize data across mobile phones and to a backend server. It has sophisticated access control, handles concurrent updates and conflict resolution, and allows inserting arbitrary processing logic into the data path. While it meets the technical requirements, Oracle's Database Mobile Server is proprietary and expensive, putting it out of reach financially for many projects in the developing world, and it is difficult to customize and extend as it is not an open-source project.

Finally, there are few systems that seek to realize similar capabilities as ODK Tables. SMS has been the basis of numerous data collection tools for the developing world, including FrontlineSMS [11], RapidSMS [20], and UjU [16]. SMS has also been used for querying, such as with SMSFind [6]. These projects are generally built for large-scale deployments and require additional equipment (such as laptops) or rely on a server in the infrastructure to process messages, thereby raising the bar to deployment. FrontlineSMS has shown the viability of many use cases for SMS data. RapidAndroid [19] also uses the smartphone as a server but does not provide a general-purpose user interface to database tables.

## 5. IMPLEMENTATION

Data tables are the underlying abstraction used by ODK Tables (hereafter referred to simply as "Tables"). Different tables can be linked – thereby creating a "join" in the vocabulary of relational databases. However, we do not write queries in SQL. Queries can be initiated through the app's search box where terms are logically ANDed and applied only to the rows of that table. The same logic is applies to SMS queries directed at a table. HTML files that are used to look at a detailed and formatted view of a row's contents can access the data linked from other tables.

Tables's user interface is optimized for the smaller screens of Android devices and provides customization capabilities for users to easily configure the app for their use case. Some of these capabilities include grouping, sorting, and updating the data in the data tables (always one row at a time – larger data changes are effected on the server-side and synced to clients)..

Tables utilizes a built-in data synchronization system that leverages existing ODK tools. The synchronization protocol is built as an HTTP API as part of ODK Aggregate, taking advantage of the security and database-agnostic server storage layer. ODK Aggregate is also container-agnostic, meaning it can be deployed to the cloud or a private server.

### 5.1 Android Client

The Tables Android client allows users to create new tables, add, delete, search, edit, and scroll through data, add columns, configure data types, view graphs, set up conditional formatting, perform summary calculations on data columns, interact with the data through an SMS interface, synchronize the data up to and down from the cloud, and set up data access controls. Many features are configured only once during setup by an administrator, who configures Tables for a particular application (or use case). Tables has 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. Tables uses Android's SQLiteDB to store both settings information and user data.

Initially, the database is populated with several tables: one for table properties (*table_definitions – an entry for each table*) and one for column properties (*column_definitions – an entry for each column in every table*). The *table_definitions* table contains information about the user-created tables, and has columns for data such as the table ID and the table's name. Another table, *table_key_value_store*, holds settings that are required only for correctly displaying the table. This includes such data as display name, column order, and font size for the default spreadsheet-like view. The *column_definitions* table holds information about all the columns of each table, including the ID of the table of which the column is a member, the column ID, and the column name. Another table, *column_key_value_store*, is the column analog to *table_key_value_store*, and contains the same sort of Tables-specific information, but relating to columns. 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 a column's values are restricted to a limited number of specific strings (such as the multiple-choice questions found on many paper forms), Tables has a multiple-choice data type that includes a list of allowable data values for that 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 interactions using SMS messages.

We use CSV files as an interchange format to easily move files between any combination of servers and clients. If a user has data to import (such as from an existing Excel spreadsheet), they can put a CSV file on the device and use Tables's CSV importer to create a new table with data from the CSV file. Users can export CSVs from 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 also create an empty table with no initial columns, and construct the table by adding each column individually from within the app.

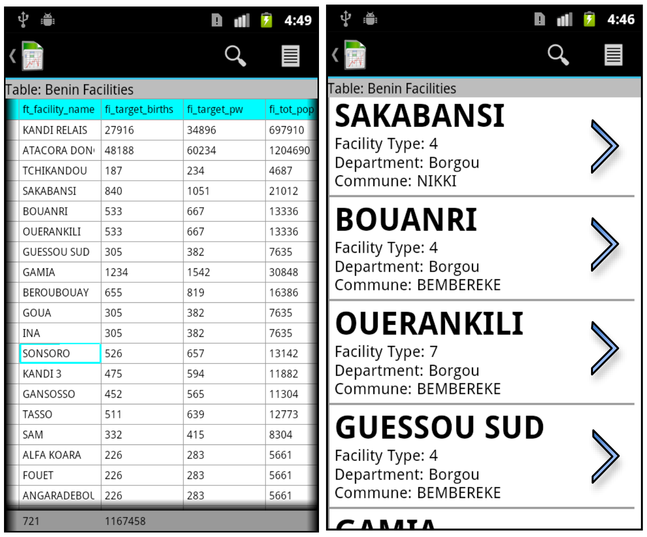
Table: Benin Facilities

| ft_facility_name | fi_target_births | fi_target_pw | fi_tot_pop |
|---|---|---|---|
| KANDI RELAIS | 27916 | 34896 | 697910 |
| ATACORA DON | 48188 | 60234 | 1204690 |
| TCHIKANDOU | 187 | 234 | 4687 |
| SAKABANSI | 840 | 1051 | 21012 |
| BOUANRI | 533 | 667 | 13336 |
| OUERANKILI | 533 | 667 | 13336 |
| GUESSOU SUD | 305 | 382 | 7635 |
| GAMIA | 1234 | 1542 | 30848 |
| BEROUBOUAY | 655 | 819 | 16386 |
| GOUA | 305 | 382 | 7635 |
| INA | 305 | 382 | 7635 |
| SONSORO | 526 | 657 | 13142 |
| KANDI 3 | 475 | 594 | 11882 |
| GANSOSSO | 452 | 565 | 11304 |
| TASSO | 511 | 639 | 12773 |
| SAM | 332 | 415 | 8304 |
| ALFA KOARA | 226 | 283 | 5661 |
| FOUET | 226 | 283 | 5661 |
| ANGARADEBOL | 226 | 283 | 5661 |
| 721 | 1167458 | | |

Table: Benin Facilities

SAKABANSI
Facility Type: 4
Department: Borgou
Commune: NIKKI

BOUANRI
Facility Type: 4
Department: Borgou
Commune: BEMBEREKE

OUERANKILI
Facility Type: 7
Department: Borgou
Commune: BEMBEREKE

GUESSOU SUD
Facility Type: 4
Department: Borgou
Commune: BEMBEREKE

**Figure 1: Table view of cold chain facilities (left) and a list view of the same information (right).**

### 5.1.1 *Views*

Tables provides a number of different data views and allows users to choose which view they prefer on a per-table basis. The default view is, naturally, the canonical spreadsheet table view, and is shown in Figure 1 (left). The colors in the middle column are an example of conditional formatting. A list view (see Figure 1 (right)) is specified by an HTML/JS file and the application developer can choose which subset of the table data to display and how.

Tables also has graphical (see Figure 2 (left)), map, and HTML views of a particular row in more detail (see Figure 2 (right)). For the graph views we use the d3.js package [25] and can have graphs configured by type, with x and y axes chosen from the colmns in a table. The detailed view is another HTML/JS file that can also display information from lined tables. The map view uses the Google Maps API (not shown) including off-line caching. Colors from conditional formatting are passed down to all these views so that the user sees a consistent formatting independent of view.

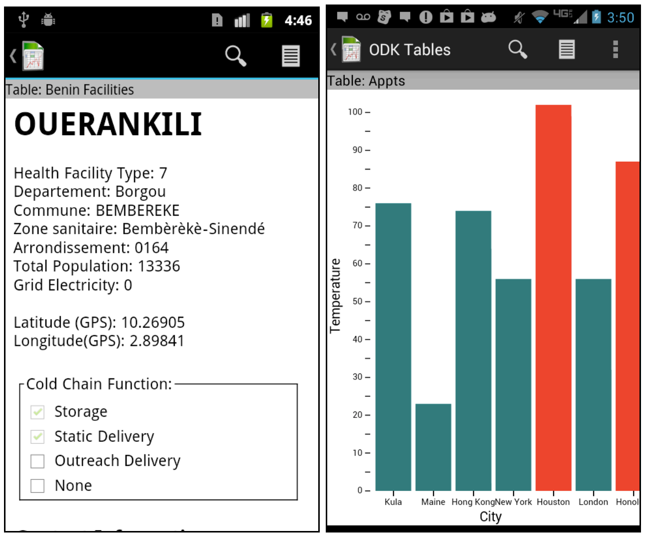
Table: Benin Facilities

OUERANKILI

Health Facility Type: 7
Departement: Borgou
Commune: BEMBEREKE
Zone sanitaire: Bembèrèkè-Sinendé
Arrondissement: 0164
Total Population: 13336
Grid Electricity: 0

Latitude (GPS): 10.26905
Longitude(GPS): 2.89841

Cold Chain Function:
✓ Storage
✓ Static Delivery
☐ Outreach Delivery
☐ None

**Figure 2: A detailed row view of one of the facilities of Figure 1 (left) and a graph view (right) including conditional formatting from a different table.**

There can be multiple HTML/JS files for list and detail views which the user can select. The HTML/JS files can be transferred directly onto the device from any connected computer, or can be uploaded to the server and synched to the phone. Tables provides templates that require simple JS commands for accessing data and basic HTML for formatting. This is accomplished through two objects ("data" and "control") passed from the Java environment of the app to the JS environment of Android WebViews that is used for rendering. With these objects, the user's custom view can access data and open new views. Additionally, the data object allows access to data by row number and column name. The control object can be used to query for additional information (from any table), with the same user search format as used in the app's search box.

### 5.1.2 *Adding, Browsing, and Searching Data*

Tables utilizes ODK Collect to enter and edit data. Collect is based on a simple form model but support rich data types including locations, barcodes, photos, audio recordings, etc. Double clicking on a cell initiates an Android Intent to Collect along with a dynamically generated form with just a single entry for that cell. This approach allows editing to be done in an enforced format (set by the column properties) rather than in free-form text which is highly error-prone. For example, a column providing four options would obtain a Collect form with those four options listed with a radio button showing the current value. Large data types such as photos have the filename and path entered into the cell so that a current file can be found an displayed or a new file created by firing an Intent to the camera.

Entire rows can be edited similarly through a dynamically generated multi-screen form, or through an optimized form to accompany the table that is prepared in advance. In fact, Tables even supports a column data type of "form" so that individual rows can trigger different forms (an example of this would be multiple medical forms to be completed by different patients visited by a community health worker). Thus, users can launch ODK Collect with a form using current values in a row as starting values. On returning to Tables from Collect, the values from the ODK Collect instance fields that match table column names replace those in that row.

When viewing data in a large table, we often want to organize it 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, they would likely want to be able to see: a) the most recent data for all refrigerators and b) all data for a particular refrigerator. Relational databases do this with "GROUP BY" and "ORDER BY" statements in SQL. Tables can designate columns as "index columns," meaning they are used to group data or "collections". 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 particular group. A user can also designate a "sort column," which is used to determine which row from each group is shown in the overview – a combination of "ORDER BY" and "FIRST()" in SQL – (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). 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 (logically ANDed) and obtain a subset of the table that matches the search terms. All views work the same way, with only items matching the search terms currently in the search box being listed, graphed, or mapped. In essence, we map some of the most common SQL constructs in simple column properties and search conditions.

To perform a simple search a user puts the column name in front of the search text. For example, to search patient records for Bob Smithson the search box would contain the string "name:Bob Smithson". To narrow the search for a tuberculosis patient the string would be revised to "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:mydistrict 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:mydistrict"* restricts the results to clinics in mydistrict. The user's search results are the clinics in mydistrict that have a spare condenser. We are working on simplifying this syntax.

## 5.2 Synchronization Protocol

The synchronization protocol provides a mechanism to keep data on multiple devices running Tables synchronized to a master copy stored in the cloud. For our initial implementation we built the data synchronization cloud service into ODK Aggregate to leverage the security and database agnostic storage layer, which enables operation from the cloud or a private server. Aggregate hosts the master table and is used to synchronize multiple instances of Tables running on client devices. The basic architecture shown in Figure 5 consists of an HTTP API exposed by Aggregate that is contacted by one or more mobile devices running Tables to synchronize data. The API is based on a REST architecture and defines a set of resources that can be manipulated using common HTTP verbs such as GET, PUT, and DELETE. The top-level resource of the HTTP API is a table. Under this are the sub-resources rows, columns, properties, and access control list (ACL). Authentication is handled using Google accounts and the OAuth2 [17] protocol.

All operations are idempotent, thereby simplifying the API because clients do not have to worry about losing results, as requests can be safely repeated. Concurrent access to the rows of a table are handled with a form of optimistic concurrency control, where once a set of changes are accepted then all others will be rejected until the client synchronizes to the currently accepted changes. When a Tables instance makes a request to update a row in a table in Aggregate, Aggregate locks the entire table and executes the update, which includes updating the version of the row, called the "entity tag". Any concurrent updates must wait for the lock, and when the lock is obtained and the second update attempts to alter the same row an error will be thrown because the second update's row version (i.e. "entity tag" or "etag") no longer matches the version of the master row in Aggregate. The second update will consequently be rejected, forcing Tables to resolve the conflict between the pending changes on the client and the updated master row on the server before any further changes to the master row is allowed.

A challenge in building the synchronization protocol was designing Tables to track the synchronization state for every row of a table. To simplify the many possible states in which data could exist, we created a simple a finite state machine for the life cycle of a row. The state machine is shown in Figure 3 has six possible states for a row:

- *Null:* the row does not exist
- *Inserting:* the row needs to be inserted on the server
- *Rest:* no changes to the row need to be communicated to the server
- *Updating:* the row needs to be updated on the server
- *Deleting:* the row needs to be deleted from the server
- *Conflicting:* both the Tables and the server's copy of the row have changed and the user needs to resolve the conflict between them

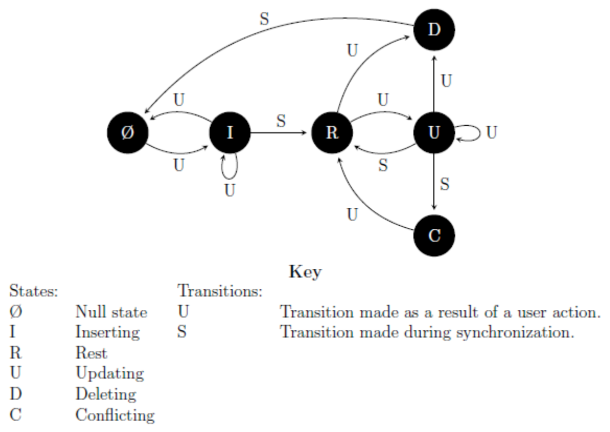

**Figure 3: Synchronization state machine for a row in Tables.**

Each transition in the state machine is either the result of a user action or the synchronization process. For example, when a user creates a new row on the phone, the row is moved from the Null state, meaning it doesn't exist, to the Inserting state. From the Inserting state, there are three different possibilities: the user deletes the row, the user updates the row, or the synchronization process runs. The row stays in the same state if it updated locally including possible deletion. If the synchronization process is run, then the row will be inserted into the table on the server and the synchronization process will transition the row to the Rest state to indicate all of its changes have been propagated to the server.

### 5.2.1 *Conflict Resolution*

Tables is designed to have the user determine how to resolve conflicts that occur whenever two users concurrently update the same row in a table. There are two main options for conflict resolution strategies: automatic or manual resolution, and server-side or client-side resolution. Manual, client-side conflict resolution was chosen for the synchronization protocol because of the complexity of trying to automatically resolve conflicts from devices that may not be time-synchronized and may be disconnected from the network for an extended period of time, making it difficult to resolve conflicts based on techniques such as most recent modification. In an automatic resolution scheme, an administrator would need to configure rules, such as to always take the changes of a certain user or group over another, or to try and intelligently merge the changes on a column by column basis. This would make the server complex to implement and would still likely lack additional configuration possibilities that organizations might desire for their application. Additionally, the assumptions built into automatic resolution may not be accurate for the many diverse use cases that exist in developing regions. The aim for the client-side conflict resolution system is to keep the user involved with reconciling conflicts. This is advantageous since many times the user understands the semantics of the conflict and can better fix the conflict than can an administrator at a later date.

To minimize the number of conflicts, updates are row-based to keep changes to small bundles. Taking cold chain inventory updates as an example, if updates are at a coarse granularity, such as the whole table or file, a conflict might be detected for two workers updating the number of fridges at different sites that do not overlap. By keeping conflict detection at the row level, multiple users can make updates to shared data and the system will detect that there is not a conflict as long as the same piece of shared data doesn't change. Cell-based conflicts would be an even finer grain unit that would reduce conflict detection further. However, in a single row many cell values are often inter-related. We felt that too much context would be lost, and that errors could occur in reconciliation due to lack of context. By always keeping the server in a consistent state, there can only ever be conflicts between a row on the user's phone and a row on the server. Furthermore, the user who caused the conflict is likely to know how to resolve it, so putting the responsibility in that user's hands is a sensible approach.

Figure 4 shows an example Tables synchronizing an existing table with Aggregate. In this example, the server has three rows with changes that Tables does not know about, and Tables has a row to insert and a row to update that the server does not know about. First, the Tables client retrieves the latest entity tags that represent the current version number for both data and properties of the table. Entity tags are universally unique identifiers to prevent multiple Tables instances from having their version identifier collide without realizing the values are different. For the purposes of this example, the versions are simply integers that are incremented every time a change is made to the table data or properties, respectively. By retrieving the latest version information, Tables can determine whether a change in the data or metadata has occurred since the last synchronization, as the version information would be different if there was change. For this scenario, Tables starts at a data version of 2 and a properties version of 2. Since the server is at a data version of 5, the Tables instance requests a diff representing all changes to the rows of the table since the table was at data version 2. The server responds with a list of the three rows that have changed. Tables then determines if the row is a new row or an update to a row it already has and inserts or updates the row accordingly. With Tables data being up-to-date with the server it is able to push its changes to the server. The server responds with the latest state of the row, including a new row version, which Tables saves with its local row copy.

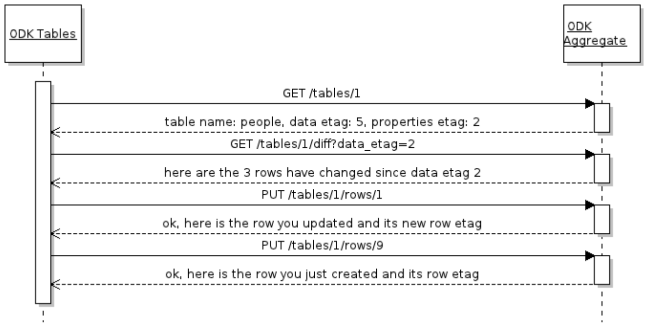
ODK Tables
ODK Aggregate
GET /tables/1
table name: people, data etag: 5, properties etag: 2
GET /tables/1/diff?data_etag=2
here are the 3 rows have changed since data etag 2
PUT /tables/1/rows/1
ok, here is the row you updated and its new row etag
PUT /tables/1/rows/9
ok, here is the row you just created and its row etag

**Figure 4: Tables synchronization sequence with Aggregate.**

### 5.2.2 *Access Control Model*

Access control is important for applications that value security and privacy, such as medical applications. Organizations might want to limit health worker access to patient data to only that worker's assigned patients. Additionally, in some scenarios managers need to be able to distribute portions of the dataset and restrict access so that field workers only need to edit/view a small subset of a dataset that is relevant to their task. The cost to distribute the entire dataset to a mobile device may be unnecessary if the worker does not need the data. Therefore, the goal is to make a system that allows administrators to control access on both coarse and fine-grained levels (table and row level), and that is generic enough for multiple usage scenarios.

When a user accesses a table, their actions are restricted based on the ACLs that have been defined. Access control at the table level is achieved by defining TableAcl resources consisting of a scope and a role. A scope defines which users the rule applies to, and the role defines the kind of access the ACL is granting. For each ACL whose scope includes a specific user, that user is conferred the permissions of the role associated with that ACL. That user's effective permissions, then, are the union of permissions from all roles which apply to them. Aside from table-level access control, the access control model also allows for row-level control. This is achieved through a *filterScope* property assigned to each row (initially set to null). Users with filtered access to the table who fall within the *filterScope* are able to access the row while users without permission will not be able to transfer the row to their mobile device. For example, if the user is a Filtered Reader on the table, and the row has a *filterScope* of Default, then the user will be able to read the row because a) the user has filtered read access to the table due to their role and b) the Default scope makes the row accessible to anyone. If, however, the *filterScope* was set to a specific user id of some other user, then the example user would not be able to access the row because they would not fall within the scope of the filter condition.

## 6. PERFORMANCE

Data set size limits Tables's overall performance because of the time needed to retrieve and render a table. Originally, the table view was implemented using Android's TableLayout; however, the UI was slow for large tables as an object had to be constructed for each cell, causing the display time to grow with the size of the table. Therefore, we created our own custom table view that scales better with varying table sizes. Since we pass large data objects between Java and WebViews and then render them with Javascript, this was another area of performance concern. Both of these aspects of rendering views are critically important because they affect the user interface and responsiveness of the app.

Obviously, the time it takes to load a table is affected by the size of the table, comprised of the product of the number of rows and columns. A two column, one row table takes approximately one second to load. A 700 row, 60 column table takes less than three seconds to load, and a 1400 row, 42 column table takes less than four seconds. Once loaded, the view is highly responsive to being scrolled and selections can be made immediately because the computation cost is proportional to what is displayed rather than the underlying table size. Operations such as queries are still proportional to the size of the table but are not tied to the display and rendering. Thus, querying the 1400 row table and restricting its contents to a single entity takes less than one second.

The two large tables in these tests were constructed from real-world vaccine cold chain data from a developing country. As performance is highly variable between mobile devices, these heuristics are not intended as true benchmarks. They are provided only to demonstrate that relatively large tables are quite feasibly managed by the platform. Further, several improvements to performance are within reach under future revisions, but have not yet been implemented. In practice, we rarely see larger tables. To date, none of our motivating use cases have ever required more than 100K cells (our largest experience is with a master table that has 1400 rows and 42 columns but with only 100-200 rows being synchronized to any one client).

As we start to investigate using devices with much larger screens such as tablets, then we will need to re-measure to determine if their typically faster processors and memory systems will compensate for the additional omputation to render the now more expansive views wth more elements showing at any one time.

Graph and map views are more forgiving in terms of required performance as they are a shift in view mode for which the user is prepared to wait an extra second or two. In any case, we are dependent on others' code for both. For 1000-2000 data rows, however, we see responses on the order of 1-2 seconds for graphs generated by d3. For maps the variability is much higher due to obtaining the map tiles, however, if the tiles are present in the cache, performance is similar to the Google Maps app.

To test the performance and scalability of the synchronization protocol we varied the following parameters: 1) the size of a table (number of rows) and 2) the number of concurrent accesses to a single table. Since the server is the center of the synchronization protocol architecture it is a possible bottleneck in the system, therefore a multithreaded test harness was created to make HTTP API calls and load the server. The tests were run using ODK Aggregate on Google App Engine (Frontend instance class of F1 - 600 MHz and 128MB). To understand how the synchronization protocol scales with an increasing table size, a test was run simulating a single user synchronously creating a table, adding a number of rows to it, performing one update to each row, then deleting the table. This was performed multiple times for tables with 10 rows, 100 rows, and 1000 rows.

Figure 5 shows the run times for the table size tests while Figure 6 shows the counts of different database operations. Both the run time and total database operations grew linearly with the table size, which makes sense because adding or updating a row should be an approximately constant time operation. This means that the scalability of the dataset size, in the case of a single user, is mostly determined by what the underlying database can handle.

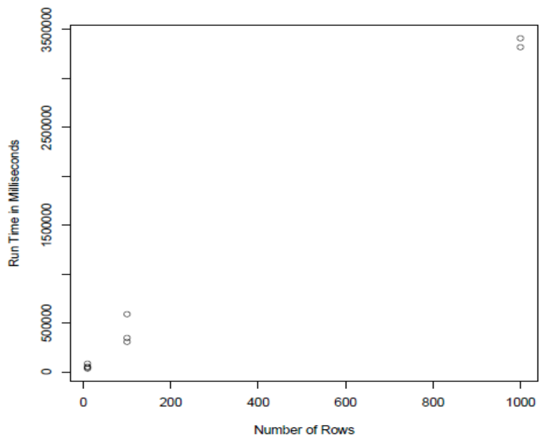

**Figure 5: Run times (ms) for table size tests.**

The second aspect of the synchronization protocol we tested was to see if it scales with concurrent access to the same table. In this test, a number of users were simulated trying to concurrently put five rows each into the same master table. Since the whole table must be locked on each update, this creates a lot of contention for the lock. On App Engine, requests are only allowed to take a certain amount of time, between about 30 and 60 seconds, and because of this, the simulated users' requests often timed out waiting for the lock. Therefore a simple exponential back-off scheme was used to retry each request until it was successful. This test was run with 1, 10, 20, 30, 40, and 100 simulated users. Figure 7 shows the run time scaled close to linearly as the number of users increased.

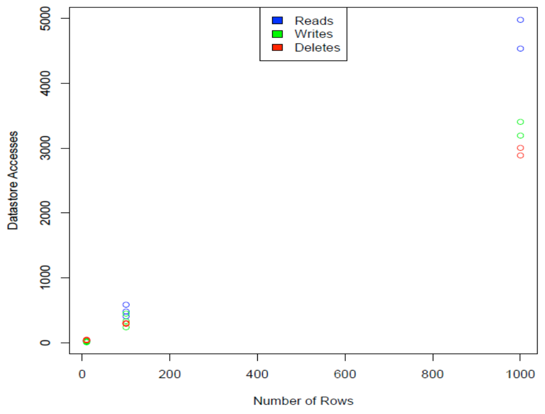

**Figure 6: Database operation counts for table size tests.**

After about 30 concurrent users the number of lock and request timeouts was fairly high, to the point where it would be noticeable to a user trying to synchronize with the server. However, if the user does not request an immediate sync and instead lets the synchronization process on run in the background at the discretion of the Tables application then it should remain unnoticeable. Overall, the synchronization protocol met the scalability as it was able to handle thousands of rows and hundreds of phones sharing a single table. For hundreds of mobile devices accessing a single table, performance depends on how many concurrent updates to a table are expected. In a deployment of five hundred phones there may only be ten simultaneous updates to the same table except possibly immediately after a service outage as everyone's device may try to synchronize within a small time window. In the multiple concurrent users tests, ten simultaneous updates were resolved within 10 seconds. The test included the time to create and delete the table, so the real time would be a bit shorter. However, because of the table lock on each update, the synchronization protocol does not perform as well for highly concurrent access. Recent changes to the Google AppEngine API make this much less of a concern as the API is changing to avoid the need for these large grain-size locks. We do not believe this will be an issue for virtually all use cases but it is something to consider during deployment scale-up.

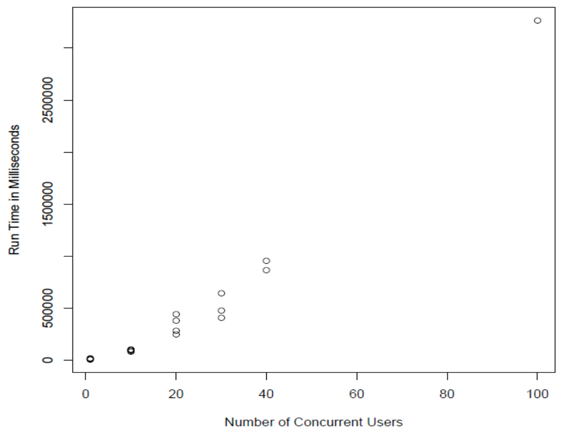

**Figure 7: Run times (ms) for multiple concurrent users tests.**

## 7. DISCUSSION AND FUTURE WORK

Building information systems in developing regions can be challenging because of the limited and diverse infrastructure available in different deployment areas. Tables aims to make it easier to build information systems that bring together different elements of technology by trying to provide a simple common format to represent data: a row in a table. A single row expression is easier to interact with over SMS than a structured document (e.g. XML, JSON) because hierarchy and nesting are not a concern and can easily be transformed into many export formats (e.g., CSV). A simple row is also generally easier for a human to visually parse.

Tables aims to present interoperability interfaces so that other specialized phone apps can be easily developed to modify data managed by Tables. Currently, Tables connects to ODK Collect (the XForm-based data collection tool of Open Data Kit), enabling new data to be entered in a structured survey/form-based interaction. Tables connects to ODK Aggregate (the database storage component of Open Data Kit) enabling data to be moved to cloud servers for backup and/or forwarded to other services, including synchronization with other mobile devices. Tables provides a local copy of cloud data on a mobile phone network opening up data access to a large number of users who only have access to basic SMS phones. This allows Tables to provide services that can be accessed from the cheapest and most common phones with the server running on a specialized mobile device (e.g, smartphone) rather than a web server.

Tables builds upon the existing ODK tool suite by enabling organizations to better customize the deployment model based on their resources and personnel. While Tables has lowered some barriers to creating information systems, there are still a number of challenges that are related to understanding user intent. Tables is written using conventional programming languages (e.g., Java, SQL) that rely on getting the exact syntax correct. Although the user interface minimizes some of these constraints, there are still times where it could become an issue. For example, because the custom HTML views and table and column settings are edited separately, it is possible for the two to become inconsistent based on user error. 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, Tables was designed by native English speakers and only offers an English user interface. It also uses American standards for date and time formatting and includes English words as syntax (for instance, "now" is recognized as input for date columns). In future versions, Tables will be 1) easy for a user to express their intentions but not overly strict on syntax mistakes and 2) multilingual and multicultural.

Additionally, a user's ability to conform to a specific syntax for SMS messages or table queries is likely to be an obstacle to using SMS for querying and adding rows. Our earlier work focused on this usage model for a previous version of Tables [14]. Rigid formats for short messages have often been problematic for on-the-ground users. For example, despite regular tweets about the syntax, a video explaining it, and a web-based tool for building tweets that conformed to the syntax, the Tweak the Tweet tool for disaster relief saw little use from the on-the-ground users for whom it was intended [21]. While Tables allows users to specify their own formats, which can be specific to the context and more natural, SMS messages for Tables still must match those formats exactly. It would be beneficial for the SMS parser to tolerate some variation, such as a typo in a column name. We also need to improve the error messages generated by Tables after receiving an invalid message. Additionally, the SMS interface permits only one operation per message, which might be cost prohibitive for large amounts of data.

## 8. CONCLUSION

With smartphones rapidly gaining adoption in the developing world there is more potential for new ways to leverage technology in ICTD work. ODK Tables occupies a design point between spreadsheets and relational databases, borrowing the best elements of each and omitting more general features that would lead to a complex user interface. It is aimed to help organizations that need to have customized information applications in contexts that necessitate a workforce with more limited technical knowledge.

ODK Tables presents a user interface optimized for the smaller screens of mobile devices, and provides customization capabilities for users to easily configure the app for their use case. Users can explore their tabular data through a variety of views enabled by a collection of HTML/JavaScript files thus making customization easy and flexible without requiring recompilation. ODK Tables enables users to enter and curate tabular data on Android devices and allows for expression of relations between datasets enabling cross-indexed data. Tables's performance results show it is well suited for scenarios with moderately sized datasets and deployments.

A key feature of smartphone applications in the developed world is synchronization of data between a backend server and one or more mobile devices. ODK Tables supports a simple synchronization scheme so that the data tables can be kept synchronized across a distributed workforce and backed up on cloud servers. The synchronization protocol, built on existing ODK tools, allows for sharing and synchronizing tables of data between mobile devices. The conflict resolution scheme was designed to involve the user in reconciling conflicts, since the user usually understands the semantics of the conflict and can more easily and accurately resolve it.

ODK Tables is designed to be a flexible information services solution for a variety of use cases including logistics management, public health, and environment monitoring where previously collected data is often revisited and updated. To determine a general set of requirements and the common features needed for Tables we analyzed five representative developing world use cases. The flexibility it provides for accessing and rendering data creates a platform that organizations can extend to implement customized information services for a variety of use cases.

## 9. ACKNOWLEDGMENTS

We gratefully acknowledge the support of Google Research, NSF Grant No. IIS-1111433, and an NSF Graduate Research Fellowship under Grant No. DGE-0718124. Nathan Brandes, Chris Gelon, YoonSung Hong, June Lee, and Heidi So made important contributions to the Tables code base. In addition to contributing to the project, Mitch Sundt provided invaluable coordination and supervision.

## REFERENCES

[1] R. Anderson, J. Lloyd, and S. Newland. "Software for national level vaccine cold chain management." In *Proc. of the Fifth International Conference on Information and Communication Technologies and Development* (ICTD '12).

[2] R.E. Anderson, A. Poon, C. Lustig, W. Brunette, G. Borriello, B.E. Kolko. "Building a transportation information system using only GPS and basic SMS infrastructure," In Proc of *Information and Communication Technologies & Development (ICTD 09),* Apr 2009

[3] Atom Syndication Format. http://www.ietf.org/rfc/rfc4287.txt

[4] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI replication," In *Proc. of the 3rd conference on Networked Systems Design& Implementation - Volume 3* (NSDI'06).

[5] R. Chaudhri, G. Borriello, R. Anderson, S. McGuire, E. O'Rourke. "FoneAstra: Enabling Remote Monitoring of Vaccine Cold-Chains Using Commodity Mobile Phones", *ACM 1st Annual Symposium on Computing for Development (DEV '10)*, Dec 2010.

[6] J. Chen, L. Subramanian, and E. Brewer. "SMS-Based Web Search for Lowend Mobile Devices"*. In Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds (MobiHeld '09).* Sept 2010. 20-24.

[7] ChildCount. http://www.childcount.org/.

[8] B. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan. "Mobius: unified messaging and data serving for mobile apps." In *Proc. of the 10th Int .Conference on Mobile systems, applications, and services* (MobiSys '12).

[9] CouchDB. http://couchdb.apache.org/

[10] Dropbox. https://www.dropbox.com/

[11] FrontlineSMS. http://www.frontlinesms.com/

[12] Google Fusion Table. http://www.google.com/fusiontables/Home/

[13] C. Hartung, Y. Anokwa, W. Brunette, A. Lerer, C. Tseng, G. Borriello. "Open Data Kit: Building Information Services for Developing Regions." In Proc. of 4th IEEE/ACM *Information & Communication Technologies for Development (ICTD 10),* Dec 2010.

[14] Y. Hong, H. Worden, G. Borriello. ODK Tables: Data Organization and Information Services on a Smartphone. 5th ACM Workshop on Networked Systems for Developing Regions (with MobiSys'11), Bethesda, MD, June, 2011.

[15] J. J. Kistler and M. Satyanarayanan. "Disconnected operation in the Coda File System." *ACM Trans. Comput. Syst.* 10, 1 (February 1992), 3-25.

[16] W. Lu, M. Tierney, J. Chen, F. Kazi, A. Hubard, J. G. Pasquel, L. Subramanian, and B. Rao. "Uju: SMS-Based Applications Made Easy." ACM 1st Annual Symposium on Computing for Development (DEV), Dec 2010. 17-18.

[17] OAuth. http://oauth.net/2/

[18] Oracle Database Mobile Server. http://www.oracle.com/technetwork/products/database-mobile-server/overview/index.html

[19] RapidAndroid, http://www.rapidsms.org/overview/projects/rapid-android/

[20] RapidSMS, http://www.rapidsms.org/

[21] K. Starbird and L. Palen. "'Voluntweeters': Self-Organizing by Digital Volunteers in Times of Crisis." In Proceedings of Human factors in Computing systems (CHI '11), May 2011.

[22] Tablecast. http://tablecast.org/

[23] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. "Managing update conflicts in Bayou, a weakly connected replicated storage system." In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (SOSP '95).

[24] R. Veeraaghavan, N. Yasodhar, and K. Toyama. "Warana Unwired: Replacing PCs with Mobile Phones in a Rural Sugarcane Cooperative", *2nd International Conference on ICTD*, 2007.

[25] d3js. http://d3js.org/

[26] Open Data Kit. http://opendatakit.org/