Google Fusion Tables: Data Management, Integration and Collaboration in the Cloud

Hector Gonzalez, Alon Halevy, Christian S. Jensen* ,
Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen
Google Inc.

ABSTRACT

Google Fusion Tables is a cloud-based service for data management and integration. Fusion Tables enables users to upload tabular data files (spreadsheets, CSV, KML), currently of up to 100MB. The system provides several ways of visualizing the data (e.g., charts, maps, and timelines) and the ability to filter and aggregate the data. It supports the integration of data from multiple sources by performing joins across tables that may belong to different users. Users can keep the data private, share it with a select set of collaborators, or make it public and thus crawlable by search engines. The discussion feature of Fusion Tables allows collaborators to conduct detailed discussions of the data at the level of tables and individual rows, columns, and cells. This paper describes the inner workings of Fusion Tables, including the storage of data in the system and the tight integration with the Google Maps infrastructure.

Categories and Subject Descriptors: H.3.5 Online Information Services: [Data sharing, Web-based services]
General Terms: Design, Algorithms
Keywords: Cloud Services, Visualization, Geo-spatial data

1. INTRODUCTION

Google Fusion Tables is a cloud-based service for data management and integration. Launched in June'09, the service (see *tables.googlelabs.com*) has since received considerable use. Although we have seen a wide range of applications, the service was originally designed for organizations that are struggling with making their data available internally and externally, and for communities of users that need to collaborate on data management across multiple enterprises.

Fusion Tables enables data upload from a variety of common sources (spreadsheets, CSV, KML, currently of up to 100MB). We make it extremely easy for users to explore their data by proposing appropriate visualizations (e.g., charts, maps, and timelines) based on the data types that are present in the data, and by enabling filtering and aggregation. Fusion Tables also provides a simple data integration platform: users can perform joins across tables that may belong to different users. To collaborate, users can share the data with a select set of collaborators, or make it public and thus crawlable by search engines. To further fuel collaboration, the discussion feature of Fusion Tables allows users to conduct detailed discussions of the data at the level of individual rows, columns or cells.

Given our target audience, we converged on a few principles that underlie the design of the service. First, our goal is to offer a tool that makes data management easier, and therefore approachable to a larger audience of users. Our target users do not necessarily have any training in using database systems, and they typically do not have access to expert DBAs. Second, it is clear that one of the main impediments to data sharing and integration was lack of *incentive*. Hence, we offer several mechanisms that provide incentives for data sharing. Third, we focus on exploring and supporting features that enable users to *collaborate* effectively on data management in the cloud. In particular, when data is shared among multiple collaborators, querying is only one part of the activity, and the system needs to support the process of agreeing on the meaning of data, including discussions on the possible errors it may contain.

This paper describes the inner workings of Fusion Tables. We describe the user-facing features and the motivations underlying them in more detail in [3].

Section 2 describes the overall system architecture and how data is stored. Section 3 covers the processing of queries, and Section 4 briefly describes the handling of transactions. Section 5 describes visualizations, and Section 6 describes the support for geographical features. We conclude with a discussion of related work and future outlook.

2. SYSTEM ARCHITECTURE AND DATA STORAGE

We begin with a brief coverage of the overall system architecture of Fusion Tables. We then consider in more detail the storage of data in the system. We briefly describe the elements of the storage stack on which Fusion Tables is built (Bigtable and Megastore), and then describe how we store all data rows of all user tables in one Bigtable table and all user table schema information in another Bigtable table.

2.1 Architecture

Figure 1 shows the main components of the Fusion Tables Service. Requests originate from multiple sources: the Fusion Tables web site, stand alone applications that use the

*On leave from Aalborg University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...$10.00.

API, and visualizations that are embedded in other Web pages (e.g., charts). We generate layers for maps based on

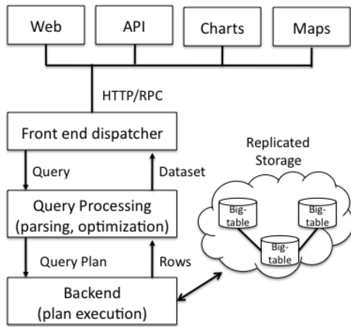

Figure 1: Architecture of Fusion Tables.

spatial/structured queries posed against tables in our system. The front end dispatcher converts requests into a common representation and passes them to the query processing module, which creates a query plan. The plan is executed by our structured data backend, which uses a set of synchronously replicated Bigtable servers for storage. The main challenge for the storage layer is the handling of hundreds of thousands of tables, with diverse schemas, sizes, and query load characteristics.

2.2 Storage Stack

Fusion Tables is built on two layers of the Google storage stack.

Bigtable: The tuples stored by Bigtable are (key, value)-pairs that are sorted on the key and sharded among multiple servers based on key ranges. As we shall see, the value part of a Bigtable tuple is allowed to be a complex non-first normal form value.

Bigtable provides a write operation that inserts a new tuple atomically. Next, BigTable provides three read operations: lookup by key, which retrieves a single pair with the given key; lookup by key prefix, which retrieves all pairs with the given key prefix; and lookup by key range, which retrieves all rows between a start and and end key.

In addition, Bigtable records the transaction-time history for each tuple. That is, internally a tuple is stored as a (key, value, timestamp)-triple, where the timestamp is the time at which the tuple was written. In the general case, a single key value is used in multiple tuple versions.

Megastore: Megastore is a library on top of Bigtable. It provides higher level primitives such as consistent secondary indexes, multi-row transactions, and consistent replication. We use this library for (i) maintaining property indexes (Section 3), (ii) providing table level transactions (Section 4), and (iii) replicating tables across multiple data centers.

2.3 Row Store

All rows in all user tables are stored in a single Bigtable table, called *Rows*. Each row in this table represents a row in a user table. The key of a row is the concatenation of identifiers for the user's table and row being represented. We generate table and row identifiers internally, as users are not required to provide a primary key.

The decision to store all tables in a single table is driven by Bigtable's excellent scalability characteristics. As new user tables are added to the system, Bigtable splits the *Rows* table into small sub-tables that are assigned to separate servers, thus providing good performance even in the presence of millions of user tables.

The (complex) value of a row is a set of indexed and non-indexed properties. A property is a pair of the form (property name, property value). Unlike in traditional database systems, we store each property name repeatedly in the Bigtable rows. This design allows us to support semi-structured data [1], where multiple rows in a single table to have different sets of properties. All indexed properties are added to an index in order to facilitate efficient query processing (discussed further in Section 3). All properties are indexed by default, with the exception of properties with long string values. In this case, only a prefix of the value is indexed and the full value is stored as a non-indexed property.

Table 1 presents an example subset of the Rows table. The example contains rows for two tables: 123 and 124. The first

Row Key (table,Id,rowId)	**Indexed Properties**	**Non-indexed Properties**
(123, 1)	model=328i, color=red, type=sedan	notes=sells quickly
(123, 2)	model=330i, color=red	
(124, 1)	price=20, location=warehouse, UPC=500	
(124, 2)	price=32, location=shelf, UPC=430	notes=reorder needed
...	...	...

Table 1: Rows table.

row, in table 123, has a value that contains properties for model, color, type, and notes. The second row, also in table 123, has a value that contains the same properties except for type and notes.

Property values are stored as strings by default, but can also contain a typed value. The typed value is automatically computed by an annotation service. Recognized types include date, number, and geo (point, line, or polygon). Types are not enforced and are only computed on a best effort basis. Types are used as a hint to the system to decide the range of available visualizations for a given table or query. For example, the string value "1600 Amphitheatre Parkway, Mountain View, CA" will be annotated as a latitude, longitude pair (37.4217760, -122.0846650).

2.4 Schema Store

The schemas of all user tables are stored in a single Bigtable table that has one row per user table. The key is the table identifier.

The value (again complex) contains column and permission information for a user table. For each column, we store its name and a preferred data type. We use access control lists (ACLs) for permissions. For each table, we list the set of users that are viewers (read permission) and collaborators (read/write permissions). Public tables have a special marker indicating that they are viewable by anyone.

A table's schema can evolve over time, users can add and remove columns, and column types (initially identified automatically) can be changed by the user. Fusion Tables is especially good at handling sparse tables, as each row stores only the set of properties that are defined for it. And typed

values are only stored if they are defined for the particular property value.

Table 2 presents the schema information corresponding to the data rows shown in Table 1.

Table	Schema	Permissions
123	name: car, columns: (model,string) (color, string) (type, string) ...	Viewers: (arl, jayant) collaborators: (hagonzal, halevy)
124	name: product, columns: (price, number) (location, string) (upc, number)	public

Table 2: Schema table.

2.5 View Store

One of the main features of Fusion Tables is that it allows multiple users to merge their tables into one, even if they do not belong to the same organization or were not aware of each other when they created the tables. A table constructed by merging (by means of equi-joins) multiple base tables is a view. Views are not materialized, and only their definition and permission information are stored. Views have their own permissions with the same roles as a table, and with an additional role: contributor. A contributor is a user that can modify the definition of the view.

2.6 Comment Store

To enable collaboration, Fusion Tables allows users to comment on tables, rows, columns, and cells. We store all the comments for all user tables in a single Bigtable table. The key of the comments table is the subject of the comment, which is the triple: (table, row, column). It uniquely identifies the element a comment applies to. The value of a row is the text of the comment, the author, and the date the comment was posted.

3. QUERY PROCESSING

Fusion Tables uses the primitives of Bigtable to support a subset of SQL queries. We currently support selections, aggregations and joins on primary keys. The general query execution strategy is to map a high-level query into the three basic operations offered by Bigtable: key lookup, prefix scan, and range scan.

We make use of a *property index* (maintained by Megastore) that speeds up a wide range of queries. The index is a Bigtable table that contains only keys, no values. The key of each row is the concatenation of a table identifier, a property name, a property value, and a row identifier. Bigtable uses key prefix compression, which is especially effective in compressing the index where entries for the same property value share the complete prefix up to the row identifier.

Table 3 presents a fragment of the property index for the rows in Table 1. Each row in the example corresponds to a single Bigtable key where the columns are concatenated.

table	prop. name	prop. value	row id
123	color	red	1
123	color	red	2
123	model	328i	1
123	model	330i	2
124	location	warehouse	1
124	location	shelf	2
124	price	20	2
124	price	32	1
...	...	...	...

Table 3: Property index.

We proceed to describe a few common query plans in order to convey the flavor of query processing in Fusion Tables. For more background on database query processing, the reader is referred to [6].

Prefix scan: Used for queries such as "select * from 123 limit 100." This is the most common type of query, as it corresponds to the default view on a table. The strategy is to do a prefix scan on the Rows table with prefix = 123.

Index prefix scan: Used for queries such as "select * from 123 where color = 'red'." The strategy is to perform a prefix scan on the property index with prefix = (123, color, red) to find the relevant rows and then retrieve them from the Rows table. In case of multiple conditions we have two execution strategies. If the indexes (for each property) are small, we scan all indexes in parallel and the intersect the results. Otherwise, we scan the indexes (for each property) sequentially, passing valid row keys from the previous scan to the next, and thus reducing the number of index entries scanned in each iteration.

Index range scan: Used for queries such as "select * from 124 where price 10 and price 20." The strategy is to do a range scan on the property index with the start key (124, price, 10) the end key (124, price, 20), and then retrieve the relevant rows from the Rows table.

Index join: Used for queries such as "select * from A, B where A.key = B.key." This is the typical view resulting from the merging of base tables. We have two basic strategies to answer the query. If one of the tables is small, we lookup each of its keys in the second table. Otherwise, we do an index merge join. We do a simultaneous index prefix scan with prefixes (A, key) and (B, key) and compute the pairs of rows that match. The pairs are then retrieved from the Rows table.

4. TRANSACTIONS

Fusion Tables is designed to be a data management platform with emphasis on collaboration and visualization of structured data. It is not designed to be a high throughput transaction processing system. Having said this, we provide transaction support for operations on a single table by using Megastore [2, 4] transaction primitives. Megastore's implementation of transactions uses write-ahead logging and optimistic concurrency control to guarantee ACID (atomic, consistent, isolated, and durable) transaction semantics.

4.1 Write-Ahead Logging

In write-ahead logging, a transaction's updates to a table are initially written to a log and only applied to the table after the transaction has committed.

Megastore uses a single Bigtable table to record the write-ahead logs for all user tables. The Bigtable table typically contains a single row for each user table. However, there can be multiple successive Bigtable rows for a user table with large log values.

The key of a row is the table identifier. The value contains the following information: the last committed timestamp, a list of unapplied transactions, and a list of mutations. A mutation captures an update of the value of a user tuple. It

is thus a tuple of the form (transaction identifier, row key, row value), where the (row key, row value)-pair identifies the data row being changed and its value after the mutation.

Table 4 presents an example log. The first entry contains the log for table 123. The last committed transaction happened at 3:00. And there are two mutations from committed transactions that have not yet been applied; the new value of row k1 and row k3 is v1 and v3, respectively.

table	timestamp	unapplied	mutation list
123	3:00	1	(1, k1, v1) (1, k3, v3)
124	3:05	-	(4, k4, v4) (4, k1, v2)
...	...	...	...

Table 4: Write-ahead log.

4.2 Transaction Life Cycle

A transaction that accesses a user table goes through the following stages:

1. **Initialization**. Read the log record for the table. Apply mutations that are committed but not yet applied. Generate a transaction identifier, and keep track of the last committed timestamp.
2. **Work**. The transaction reads and writes rows for the table. All reads are isolated—we use the Bigtable versions to ensure that the transaction only reads rows as they were when the transaction started. Mutations are written to the log, with the transaction identifier.
3. **Commit**. The transaction reads and locks the log table row for the user table. It then checks whether another transaction has committed since it started—if so, it aborts (and is restarted). If there are no conflicts, the transaction is marked as committed.
4. **Apply**. After commit, the mutations for the transaction are applied to the user table, and the transaction is marked as having been applied.

5. DATA VISUALIZATION

One of the most powerful features of Fusion Tables is that users can visualize their data immediately after uploading it. The set of available visualizations is computed based on the data types found in the table and the types required for a particular visualization. For example, a scatter plot is available only if at least two numeric columns exist, one for the x axis and one for the y axis. Similarly, a map is available if we detect a location column, e.g., a column with street addresses, or a column with latitude and longitude values.

5.1 Visualization Infrastructure

We provide client-side visualizations through the Google Visualization API. This is a well established framework for visualizing data on the client. The visualization is rendered on the browser using Javascript or Flash, and the data required by the visualization is obtained from a data source interface. A large collection of visualizations has already been created by Google and the community[1].

Fusion Tables provides two services within the framework. First, we expose tables and views as sources of data for visualizations. We accept queries for data and return an appropriately encoded result suitable to be used in any visualization. Second, we help users configure visualizations automatically based on the data types in their tables. For example, a table with a location column and a numeric column will have an intensity map preconfigured to use the location column as the geographic information and the numeric column as the intensity.

[1] *http://code.google.com/apis/visualization/documentation/gallery.html*

5.2 Embedding

To foster collaboration among users, we enable visualizations to be published in web pages. That way, the data can appear in the natural place where other content exists. Users can copy a small fragment of Javascript code into the source of their page (e.g., a blog entry) and the visualization will be displayed there, with a live link to the data. That is, when the data is updated in Fusion Tables, the visualization is also updated. At this point we do not have versioned visualization support, i.e., a user cannot show a visualization based on data as of a particular date.

Figure 2 shows a fragment of Javascript that can be used to embed a visualization in a web page. Line 4 defines the query that is sent to Fusion Tables, in this case a listing of total salary by department. Line 13 takes the received data and renders it as a pie chart. For users who do not want to write Javascript, we also generate a gadget (small fragment of code) that can be directly embedded into a web page.

```
1:  function getData() {
2:   // Construct and send the query
3:   var url='http://tables.googlelabs.com';
4:   var sql='select dept, sum(salary) from 123';
5:           +'group by dept';
6:   var query=new google.visualization.Query(url);
7:   query.setQuery(sql);
8:   query.send(handleQueryResponse);
9:  }
10: function handleQueryResponse(response) {
11:  // Draw the visualization
12:  var data=response.getDataTable();
13:  var chart=new google.visualization.PieChart(div);
14:  chart.draw(data);
15: }
```

Figure 2: Embedded visualization code.

6. GEOGRAPHICAL FEATURES

A very popular component of Fusion Tables is the rendering of large geographic data sets. We allow users to upload tables with street addresses, points, lines, or polygons. We render these tables as map layers. The rendering is done on the server side, i.e., we send the client a collection of small images (tiles) that contain the rendered map. Figure 3 shows an example of rendering the bike trails in the San Francisco bay area that are shorter than 20 miles.

In order to render a client-side visualization, we send all the data to the client, and it is the browser that renders the visualization. That model is hard to apply when a large dataset needs to be visualized. Two main difficulties exist. First, the browser may not have enough processing power to render thousands of features in real time. Second, the transmission of a large dataset to the client may be impractical.

We first provide some necessary background on the Google Maps infrastructure and then describe how we support our geographical features.

6.1 Google Maps Infrastructure

To understand how we enable map visualizations of large data sets, some background on Google Maps is needed. The information that a user sees on Google Maps at any time is an overlay of multiple layers. E.g., the street, satellite, and

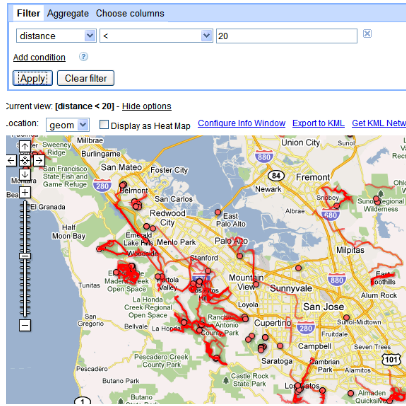

Figure 3: A visualization of all bike trails in the San Francisco Bay Area that are shorter than 20 miles.

terrain maps are separate layers. Any query result that is to be displayed on a map is represented as a layer.

When a user submits a request to view a map, a corresponding request is sent to the backend servers with information about the currently visible layers, the geographic coordinates of the window that is visible on the user's browser, and the current zoom level. The backend then creates tiles (small images) by putting together information in the different layers, and it serves the tiles as the response to the user's request.

6.2 Spatial Index

We insert the geo features in Fusion Tables into a spatial index. The index uses a space-filling curve to map points on the Earth's surface to one-dimensional values (cells).

The mapping is as follows: There are six top-level "face cells", obtained by projecting the surface of the Earth onto the six faces of a cube. Each face is then subdivided recursively into four cells in a quadtree-like fashion [8]. Thus each cell is a rectangle whose edges are aligned with the sides of the cube. On the Earth's surface, such a cell corresponds to a spherical quadrilateral bounded by four geodesics (great circle segments).

Cells are identified according to their position in the hierarchy. The total number of levels is chosen according to the desired size (square meters) of leaf cells. Any space-filling curve can be used to enumerate cells at each level. We use the Hilbert curve [7].

The left side of Figure 4 shows an example of a three-level Hilbert curve. For example, the value "5.2.3" identifies the

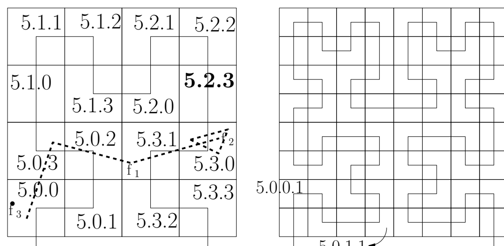

Figure 4: Examples of Hilbert curves with three and four levels.

cell that is in the sixth face of the cube (level 1), in the third cell at second level, and in the fourth cell at the third level (numbering starts at 0). The right part of the figure shows the curve when one more level has been added.

The index is a sorted list of cell identifiers. Each cell identifier points to all the features that overlap that cell. Table 5 presents and example index with three features: f_1 is a polyline, f_2 is a small polygon, and f_3 is a point. These features are also shown in the left part of Figure 4.

cell	features
5.0.0	f_1, f_2
5.0.0.1	f_3
5.0.2	f_1
5.0.3	f_1
5.3.0	f_1, f_2
5.3.1	f_1

Table 5: Example spatial index.

Inserting features: Features are inserted into the index as follows: (i) The feature is mapped to the set of cells that cover it. The cover of a feature may involve cells at multiple levels, e.g., a polygon that covers a large portion of a state may have large cells in the middle and only finer cells along the boundaries. In the previous example f_3 was covered by a cell at level 4, while the other features were covered by cells at level 3. (ii) The cells that make up the cover are inserted into or updated in the index to point to the feature.

Spatial query processing: The most common query for the spatial index is "what features fall into a bounding rectangle?" The rectangle is defined by a pair of latitude, longitude coordinates.

The strategy to answering the query is as follows: (i) Convert the bounding box into a range (or ranges) in the space filling curve. (ii) For each such range, retrieve all the features that are contained in cells between the start and the end of the range.

Spatial and structured queries: Fusion Tables supports structured queries over maps. E.g., a user may ask for all bike trails in the San Francisco Bay Area (spatial query) that have a rating of 4 stars or more (structured query).

We answer such queries by executing the spatial and structured parts in parallel and then intersect the results. The structured part is answered using the techniques described in Section 3, and the spatial part is answered using the algorithm described above.

6.3 Sampling

In the interest of ensuring fast map visualizations, a limit is placed on the number of features that can be drawn on any tile. If the number of features for a tile that satisfy a user query exceeds this limit, the Fusion Tables servers return only a sample of the items in its response to the Google Maps servers.

Sampling is done as follows. (i) We go through each feature and compute all the tiles in which it appears (at every zoom level), and we assign the feature to each such tile. (ii) The hierarchy of tiles is traversed, from low zoom (far away) to high zoom (close up). At each level, we assign a sample of features to the tile. As we go into lower zoom levels, we respect the features already assigned to the tile through parent tiles, and we add new features. This process guarantees that a tile will not have more than a predefined threshold of features, and it ensures that the sampling is consistent, meaning that points never disappear when a user moves the view port (changes the bounding rectangle) or zooms in. At the end, each feature contains just one additional attribute, the smallest zoom level at which the feature appears.

```
1: // Create a new layer for the Fusion Tables map
2: var l = new GLayer("ft:602");

3: // Draw the map as features (not as heat)
4: l.setParameter("h", "false");

5: // Display only features that match the query
6: var sql = "select col2 from 602 where length < 20";
7: l.setParameter("s", sql);
```

Table 6: Embedded map code.

6.4 Heat Maps

Fusion Tables also supports the rendering of heat maps. This is useful when a user wants to see a map colored according to the density of features in space. It can also be used to visualize very large data sets where sampling may not capture subtle differences in feature density.

We build heat maps as follows. (i) Retrieve the set of features that fall into the viewport using the spatial index. (ii) Divide the viewport into a fine grid. (iii) Count the number of features in each grid cell (iv) Color grid cells using a palette that assigns light colors to low-count cells, and strong colors to high-count cells. Cells with no features are not colored.

6.5 Embedding

As with client visualizations, maps can be published on web pages. Users can copy small fragments of Javascript code into the sources of their web pages. The corresponding maps will be displayed with a live link to the data.

Table 6 shows a fragment of Javascript code used to embed a map into a web page. Line 2 creates the layer, named ft:tableId—each table has its own layer. It is possible to add multiple tables to the same map, by just adding their respective layers. Line 4 tells Fusion Tables to draw the layer as features, and not as a heat map. Line 7 sets the SQL query that filters the set of relevant features.

7. FUSION TABLES API

An important aspect of being a platform for data management and collaboration is to provide developers with a way of extending the functionality of the platform. We accomplish this through an API.

The API allows external developers to write applications that use Fusion Tables as a database. For example, the site *mtbguru.com* has written an application that synchronizes its collection of bike routes with a table in Fusion Tables.

The API supports querying of data through select statements, modification of the data through insert, delete, and update statements, and data definition through a create table statement. We do not currently support altering table schemas through the API. All access to data through the API is authenticated through existing standards.

8. RELATED WORK

Fusion Tables is inspired in part by ManyEyes (*manyeyes.com*) that enables users to upload data and visualize it in several ways. We go further by providing data management capabilities and a sharing model that does not require that users always make their data public. There are several other online database tools such as DabbleDB (*dabbledb.com*), Socrata (*socrata.com*), and Factual (*factual.com*), but Fusion Tables focuses on the collaboration aspects of data management and handles larger data sets. In addition, Fusion Tables emphasizes the deep integration into a maps infrastructure that is proving to be immensely popular. Wolfram Alpha is a search engine for structured data, while our focus is on enabling users to manager their own data, but we will support search for public tables. SimpleDB (*aws.amazon.com/simpledb*) is a database service in the cloud but is targeted at developers.

Several projects related to structured data also exist at Google. Google Public Data is an effort to import public data and to provide high-quality and carefully-chosen visualizations of such data in search results. For example, a query on the Google search engine for "California unemployment rate" will lead the user to a page where she can explore the unemployment in California over time and compare it with other states. The Google Squared Service lets users specify categories of objects (e.g., US Presidents, espresso machines) and then explore attributes of entities in these sets. The data populating the tables is automatically extracted from various sources on the Web, and it may not always be accurate.

9. CONCLUSIONS

We described the main aspects of the architecture of Fusion Tables, a cloud-based data management service that focuses on the collaborative aspects of data management. Both the challenges we face and the advantages we gain stem from the fact that we do our best to integrate Fusion Tables with the existing Google infrastructure. Using Bigtable and Megastore provides us with a scalable and replicated data store (albeit not for very high transaction rates). On the other hand, this makes it trickier to implement certain kinds of SQL queries. The Google Maps infrastructure enables us to provide a user experience that integrates seamlessly with other map features.

Moving forward, our goal is to provide a data management experience that seamlessly integrates with other experiences on the Web. We believe that integration with the Web is a promising approach to making data management applicable to a broad set of users [5]. In addition to integration with Google Maps, this means integration with search (e.g., our public tables are already crawlable) and with enterprise collaboration tools such as Google Docs.

Acknowledgments

We thank Mike Carey for the many insightful comments on an earlier draft of the paper.

10. REFERENCES

[1] S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.

[2] J. Furman, J. S. Karlsson, J.-M. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. In *SIGMOD*, 2008.

[3] H. Gonzalez, A. Halevy, C. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google Fusion Tables: Web-Centered Data Management and Collaboration. In *SIGMOD*, 2010.

[4] J. Hamilton. Perspectices - Google Megastore. perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx.

[5] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, 2007.

[6] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2002.

[7] H. Sagan. *Space-Filling curves*. Springer-Verlag, Berlin/Heidelberg/New York, 1994.

[8] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.