# Dexter: Plugging-n-Playing with Data Sources in your Browser

**Abhijeet Mohapatra** and **Sudhir Agarwal** and **Michael Genesereth**
353 Serra Mall (Gates Bldg.), Stanford University,
Stanford CA-94305, U.S.A

## Abstract

We present *Dexter*, a browser-based, general purpose data exploration system for end-users. Dexter enables end-users to easily query across multiple Web-accessible heterogeneous (semi-) structured data sources with higher expressivity than that is usually directly supported by the sources. A novelty of our approach lies in the client-sided evaluation of end-user queries. Our query evaluation technique exploits the querying capabilities of the sources and communicates directly with the sources whenever possible. Dexter-Server, the server-sided component of Dexter, merely acts as a proxy for accessing sources that are not directly accessible from a Dexter-Client. Dexter also supports organizational internal and personal data sources while respecting end-users' security and privacy. We present the results of our evaluation of Dexter for scenarios that involve querying across data about the U.S. Congerss, the U.S. Code, and feeds from popular social networks. We discuss the applicability of Dexter for data about cities.

## Introduction

End-users often need to quickly integrate and analyze information from multiple information sources including Web sites, for example for decision making, hobby, general knowledge or simply for fun. Often such tasks are dynamic and volatile and performing them with the state of the art tools is very cumbersome, tedious, and time consuming.

Publicly available data sources usually support *limited querying capability*, because allowing arbitrary queries could potentially overload the servers, and expose the sources to DoS attacks. Consider the Web site `https://www.govtrack.us` which has information about the age, role, gender of U.S. congress persons till date. Even though Govtrack has the required data, it is still very hard to find out 'Which U.S. senators are under 40 years old?', or 'Which U.S. states have currently more female than male senators?' just to name a few examples. It is even harder to find answers to queries that require data from multiple sources. For example, 'Which congress members were Head of DARPA or Deputy Director of NASA' or 'How did the senators who like Shooting as a sport discipline vote for bills concerning gun control?' In addition to publicly accessible sources, end-users often have access to organizational internal data sources or local data in form of files or tables that are extracted from Web pages. When end-users wish to query such data sources, *privacy and confidentiality* become an important issue and data-shipping (Kossmann 2000) is not feasible.

Popular search engines do not support *compilation of information from multiple documents*, a prerequisite to satisfy the overall information need of an end-user. Popular spreadsheets do not support increasingly sophisticated data integration and query requirements of end-users such as *joins across multiple sources*. Both Semantic Web (Berners-Lee, Hendler, and Lassila 2001) and Linked Data (Bizer, Heath, and Berners-Lee 2009) rely on providers' cooperation for annotating their websites or providing semantic data. However, end-users cannot be expected to consume RDF directly. For detailed analysis of Dexter-related tools and technologies we refer to (Agarwal, Mohapatra, and Genesereth 2014).

**Running Example:** Suppose that Columbus is an end-user who is browsing the Politico website. Columbus notes down the names of the senators who voted against ending the U.S. Government shutdown of 2013 in a file (say localFile.csv). Columbus is curious to find out from Govtrack which of these senators are from Texas. After Columbus discovers the relevant Texan Republican senators, he wants to find out what these senators are tweeting about, say, the IRS scandal.

With the previous techniques, Columbus has to repeatedly access the data sources, and either (a) manually correlate file data with the data from Govtrack and Twitter or (b) upload localFile.csv into a database server to automate the answer compilation. In case (a), computing the answer is potentially very tedious and time-consuming. In case (b), Columbus compromises the *privacy* of his local data by uploading localFile.csv to a server.

## Our Contribution

We present Dexter, a domain-independent, browser-based tool that enables end-users to easily connect, integrate, and query data from multiple *plug-n-play* sources, to better exploit the data that is accessible to them. Dexter supports publicly accessible data sources such as Web-APIs and or-

ganization internal databases. In addition, Dexter can also import data from common formats such as CSV, XML, and JSON as well as allows end-users to extract interesting data from Web pages while they browse (Agarwal and Genesereth 2013). Dexter respects users' *privacy* by storing users' data locally inside their Web browsers and not on a remote server.

The end-user queries are expressed in Datalog (Garcia-Molina, Ullman, and Widom 2009) that is extended with arithmetic and logical operators, and aggregates (Mohapatra and Genesereth 2013). As a result, Dexter supports queries with *higher expressivity* (for e.g. recursive queries) than that are supported by server-sided query evaluation systems (Kossmann 2000).

A novelty of Dexter lies in the *client-sided evaluation* of end-user queries. The client-sided query evaluation in Dexter follows a hybrid-shipping strategy (Kossmann 2000) and exploits the querying capabilities of the sources. Dexter does not ship users' local data or organizational internal data to a remote server. This not only further contributes to privacy but also improves the performance of queries over sources that are directly accessible from the client by reducing the communication cost of shipping the data to a remote server.

Dexter enables end-users to query not only individual data sources but more importantly across multiple data sources. The high expressivity fills the gap between the provided and the desired view on information. The client-sided query evaluation is combined with an *intuitive user interface* to allow even the end-users with little technical background to query data from multiple sources. Dexter relies on end-users to resolve entities (Elmagarmid, Ipeirotis, and Verykios 2007) by defining appropriate views and (in)-equalities.

## Dexter: An Overview

In this section, we briefly describe how Dexter allows end-users to easily incorporate Web-accessible data sources (*plug*) and query them in ad-hoc fashion (*play*).

### Incorporating Sources

End-users can connect to, and query a diverse set of structured and semi-structured data sources with Dexter. These sources could be local files (e.g. CSV, XML, and JSON), APIs (e.g. Twitter), databases or web-pages. Dexter-Client communicates directly with directly accessible sources e.g. local files and JSONP APIs. For other sources (e.g. Twitter API), the connection between Dexter-Client and the sources is established using Dexter-Server, the server-sided component of Dexter. When end-users plug a supported type of data source into Dexter, the data is converted to the Relational Model as follows.

**Extraction from Web Pages** The conversion of an HTML DOM fragment (copied and pasted by an end-user) to a table is performed with the technique presented in (Agarwal and Genesereth 2013). The extracted tables are stored locally inside the browser of an end-user.

**Importing Files** CSV, XML, and JSON files in the local file system or accessible via a URL are handled in a similar fashion. First, the file content is fetched. Then the content is converted into a set of tables. In case of CSV file the output set contains exactly one element. In our running example, Columbus would import the file LocalFile.csv in Dexter and have obtain a table, say, `localFile`. In case of XML or JSON files, the output set may contain multiple tables since XML and JSON are nested data structures. The tables are stored locally inside the browser of an end-user.

**Connecting Web-APIs** Majority of the Web-APIs support JSON format. In contrast to traditional Web applications which hard-code the set of underlying sources, Dexter is based upon a generic source model and allows end-users to specify the source-specific information as an instantiation of the generic model at the time of plugging in a source.

The generic model of a remote source is a tuple $(name, type, direct, fnlowering, fnlifting, \mathcal{O})$, where

- *name* is the name of the source used for referring to the source in the UI and in the queries,
- *type* is either 'none', 'filter' or 'datalog', and determines the querying capability of the source, where 'filter' means that the source supports selection of rows based on provided values for fields, and 'datalog' means that the source supports a datalog query,
- *direct* is either 'true' or 'false' and determines whether a Dexter-Client can directly access the source (JSONP) or needs to use Dexter-Server as proxy,
- *fnlowering* is a function that takes the object type, an offset and a limit, and returns the arguments required for source invocation to fetch instances of the object type ranging from offset to offset+limit,
- *fnlifting* is a function that converts the returned instances to a table, and
- $\mathcal{O}$ is the set of specifications of object types. An object type $O \in \mathcal{O}$ is a tuple $(name, properties, URL)$, where *name* is the name of object type $O$, *properties* are the properties of $O$, and *URL* is the URL required to fetch instances of $O$.

Dexter stores the set of above mentioned source definition tuples inside end-user's local browser so that they need to specified only once.

### Queries over Sources

End-users can query his or her set of sources by using Dexter's UI. The end-user queries are expressed in Datalog that is extended with arithmetic and logical operators, and aggregates over the relational schemas of the plugged sources. The queries are evaluated on the client side to ensure privacy. In our running example, the tweets of Texan Republican senators can be expressed by the following query.

```
q(T) :- senator(U, "Rep", S) & tweet(U, T)
```

Of course, the join over `U` has the intended meaning only if the syntactic equality of the first column of `senator` and

`tweet` translates to semantical equality. We rely on the end-user to decide which predicates he or she wishes to join in order to have meaningful answers. When a query has been evaluated, Dexter allows an end-user to export of query results in CSV format.

In order to enable end-users to construct complex queries modularly and to support efficient evaluation of queries, Dexter allows end-users to define and materialize views over the available sources. The views are defined with the same formalism as the queries. A view definition is a set of extended Datalog rules, where each rule has the same head predicate. Note that multiple rules with the same head predicate denote union in Datalog. View definitions of an end-user are stored on the client-machine. To avoid the recomputation of the results of a view in a query, an end-user can choose to materialize a view. Materializing a view can significantly speed-up the time taken to evaluate a query that is defined over the view. If an end-user chooses to materialize a view, the evaluation of the view definition is materialized on the client side.

## Client-sided Query Evaluation

The end-user queries are expressed in Datalog over the (relational) schemas of the involved sources, and are evaluated on the client side to ensure privacy. The naive approach of first fetching all the required data to the client machine and then evaluating the query answers, also referred to as Data-Shipping (Kossmann 2000), is not practical since the client machines are usually not high-performance machines. We follow the Hybrid-Shipping strategy (Kossmann 2000) for achieving acceptable query answering performance without requiring the end-users to compromise on their privacy or overloading client machines.

The views in the query schema, some of which are potentially materialized, map to the data in the sources. Suppose that the views `senators(Name, Party, State)` and `tweets(U, Tweet)` represent the senator data from Govtrack and the Twitter posts of a user `U` respectively. The internal schema is automatically generated in Dexter when connecting to the sources. The architecture of the query evaluation process in Dexter is presented in Figure 1.

The input query is then processed in three stages. First, the query is *decomposed* into partial queries that are sent to the respective sources. For example, the input query `q(T)` is decomposed as follows.

```
q(T) :- q1(U) & q2(U, T)
q1(U) :- senator(U, "Rep", S)
q2(U, T) :- tweet(U, T)
```

The partial queries `q1(U)` and `q2(U, T)` are sent to Govtrack and Twitter respectively. The input query could, however, take too long to answer either because the partial queries are too complex or because the size of the answer is too large. Therefore, the partial queries are *fragmented* horizontally (Kossmann 2000) before sending them to the sources to avoid the bottlenecks of the query and communication times. In the final step, the answer to the input query is constructed from the answers to the query fragments.

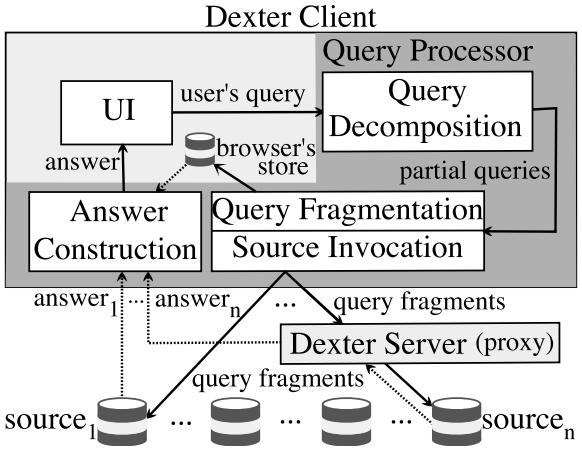


Figure 1: Query Evaluation in Dexter

**Query Decomposition** A query over the internal schema is decomposed into partial queries and rules that assembles the answers to the partial queries. The query decomposition in Dexter is a hybrid-shipping strategy (Kossmann 2000) and depends on the distribution of data to which the views in the internal schema map and the querying capability of a source. For example, suppose that Govtrack is a relational database. Since databases allow filters on any attribute of a table, the query `q1(U)` is sent to Govtrack. However, if Govtrack does not allow the `senator` table to be filtered by the attribute `Party`, then the whole table is shipped to the Dexter-Client where the filters are, subsequently, applied. In addition, if it is not known whether a certain operation (say, join or negation) can be performed at a source, then the relevant operation is performed on the client-side after shipping the data from the source. We note that, in Dexter, the queries over local data (such as CSV, XML or JSON files) are *always* evaluated on the client-side and never shipped to a source. Thus, the privacy of the local data is always preserved. For example, in the following query that finds the Texan Republican senators who voted against ending the shutdown, the data from `localFile` is *never shipped* to the Govtrack source to filter the `senator` table on the `Name` attribute in addition to the `Party` and `State` attributes.

```
q'(T) :- senator(U, "Rep", "TX") &
         localFile(U)
```

**Query Fragmentation** In Dexter, an input query is decomposed into partial queries that are shipped to the sources and the answers to the input query is constructed on the client-side from the answers to the partial queries. The evaluation of the partial queries can be bottlenecked by (a) the complexity of the query and (b) the size of the answers, especially when the size of the data that is shipped from a source is much greater than the size of the browser's local store of an end-user. To relieve these bottlenecks, the partial queries are fragmented horizontally into chunks based on the the size of the browser's local store and the maximum number of an answers to a query that can be returned from a source. For example, suppose that the Twitter API allows a

maximum of 500 tweets to be returned per call. If the number of senators and the total number of tweets are, say, 500 and 10000 respectively, then the partial query `q2(U, T)` is fragmented into 20 fragments. We note that, in addition to the partial queries, the rules that assemble the answers to the partial queries are fragmented as well to support pipelined execution of queries.

**Source Invocation** After fragmenting the partial queries, the fragments are shipped to the sources in parallel by invoking the appropriate wrappers which could be located either on client (e.g. local files, servers that support JSONP requests) or on the server (e.g. Twitter API, databases such as MySQL).

**Answer Construction** In the answer construction stage, the answers of the partial query fragments are assembled to compute the answer to the query that was supplied as an input to the query decomposition stage. The set-union of the query answers is computed for positive Datalog queries. However, for queries that involve negation or aggregation, the rules that combine the answers to the partial queries are rewritten. For example, suppose a query, say, `q(X) :- qS(X) & not qT(X)` is fragmented into multiple queries which are defined as `qij(X) :- qTi(X) & not qSj(X)` where $i$ and $j$ are bounded by the number of fragments in source S and T respectively. In this case, the answers of `q(X)` = $\{$`X` $\mid \forall i, j$ `qij(X)`$\}$.

## Implementation and ExploreGov Case Study

As mentioned before, Dexter is a domain-independent browser-based data explorer. We have evaluated Dexter for a case study that involves exploration of data about the U.S. Congress in conjunction with the U.S. Code and feeds from popular social networks. In the next section, we motivate the applicability of Dexter to explore semantic cities' data.

### Implementation

Dexter-Client is implemented in Javascript and JQuery. Coordination of parallel query processing threads (corresponding to the query fragments) is implemented with the help of Web Workers. The Answer Construction module is implemented by using the Epilog Javascript library (Genesereth 2013). We have extended the Epilog library to output answers in a streaming fashion to improve query responsiveness. Dexter-Client is equipped with a caching mechanism in order to reduce communication with and load on servers as well as to enhance client-sided query performance. Dexter-Server is implemented in PHP and Python, and acts primarily as proxy for data sources that are not directly accessible from a Dexter-Client.

### Case Study: ExploreGov

The ExploreGov case study involves exploration of data about U.S. Congress. and senators along with their biodatas and social network posts. For the ExploreGov case study we use the following sources:

1. Govtrack API for queries over latest U.S. executive and legislative data. Govtrack API can be accessed directly from Dexter-Client since it supports JSONP.
2. Govtrack Archival DB for very complex queries on archival data. We have downloaded the Govtrack Archival DB and we host it on a DB server reachable from the Dexter-Server. A Dexter-Client can access the Govtrack DB via a PHP script hosted on Dexter-Server.
3. Cornell Law for obtaining the text of U.S. Codes referred to by the bills. This API is a completely client-sided API. It requires concrete input values for the title and the section of a U.S. Code and returns the text of the U.S. Code. Therefore, it is invoked during query processing since the bindings for inputs are not known before.
4. Twitter and Facebook. PHP scripts hosted on Dexter-Server to fetch tweets or Facebook posts of persons (congresspersons, senators, representatives etc.) by using Twitter API (Twitter 2014) and Facebook API (Facebook 2014). In order to not overload the Twitter server, the Dexter-Server caches the tweets. The API interface `twitter.tweets(ID, TWEET, LINK, TIME)` also accepts constraints on the four fields in SQL syntax, e.g. `twitter.TWEET LIKE "%ObamaCare%"`. The API for Facebook posts functions analogously.
5. Further sources containing information about U.S. colleges and state relationships, general geographic and demographic information about the U.S. states, and bioguide which contains brief bio-datas of U.S. congresspersons.

**Evaluation Results:** We have evaluated many queries with varying complexities (from simple selections and projections to joins with negation and aggregation) on and across the above mentioned sources.

An example query 'U.S. presidents who attended college in Harvard University and some college in CA' required joining Govtrack data with bioguide and colleges data. Dexter required only 5 seconds to find out that Barack Obama and John F. Kennedy are the only matches for the query.

For the example query involving negation 'Which U.S. states currently do not have any male Republican senators', Dexter could find out within 15 seconds that NH is the only such state.

For the query 'Male senators who recently tweeted about equal pay' which required joins over Govtrack data and Twitter data, Dexter could find out the senator names and their relevant tweets within 10 seconds.

The query 'Which party do the maximum number of presidents come from?' required aggregates. Our system could find out the correct answer 'Republican' within 3 seconds.

We refer the reader to the online demo at `http://dexter.stanford.edu/semcities` for further tested queries along with their Datalog formulations. Furthermore, we encourage the readers to try their own queries in Dexter.

## Conclusion and Outlook

We presented Dexter, a tool that empowers end-users to *plug-n-play* with (semi-) structured data from heterogenous data sources such as databases, local files (e.g. CSV, XML and JSON), Web pages and Web-APIs. Dexter allows end-users to exploit their *small-data* in contrast to big-data analytics which is used by large organizations to draw greater value from their ever increasing data collections. Dexter-Client communicates directly with data sources when possible, and through Dexter-Server (proxy) otherwise. Dexter preserves the *privacy* of end-users' local data through a *client-sided query evaluation* strategy in which the users' local data is never shipped to a remote server. By enabling end-users to pose arbitrary queries over a source or across sources, Dexter bridges the gap between the querying capability of sources and the information need of a user.

**Exploring Semantic Cities through Dexter** The semantic city initiative is driven by open data for example `https://www.data.gov/cities/` that is made publicly available by the cities. This data is either available for download in CSV, XML or JSON formats or accessible through public JSONP APIs. Since Dexter is a domain-independent data-explorer that supports queries over the open data formats, it can be used by end-users to explore and query across the open city data. We motivate the use of Dexter to explore semantic cities' data and to build *smart services* (IBM 2014) through the following examples.

**Example 1.** Suppose Columbus is visiting Washington D.C. and is interested to find out popular local restaurants that are hygienic. We assume that Columbus has plugged the Yelp API (YELP 2014b) into Dexter and is exploring the popular Washington D.C. restaurants. In addition, we also assume that the Yelp data is queried as the predicate `yelp(NAME, ADDRESS, RATING, CATEGORY)` in Dexter.

Recently, Yelp introduced an initiative to display hygiene scores for restaurants that are reported by municipalities. A study presented in (Simon et al. 2005) suggests that there is a marked decrease in the number of hospitalizations due to food borne diseases (for e.g. 13% in Los Angeles County) if restaurant hygiene data is made available. To display the hygiene scores Yelp requires that the municipalities upload the health inspection data as per the LIVES standard (YELP 2014a). Although, there is an abundance of free publicly available hygiene data of restaurants, only the data reported by the municipalities of San Francisco and New York conform to the LIVES standard. Therefore, Yelp currently does not display the health scores of restaurants except those which are in San Francisco or New York. For instance, the results of the food safety inspections in Washington D.C. restaurants are available at (OpenDataDC 2014) as CSV files but the schema of these CSV files does not conform to the LIVES standard. This is a typical scenario in which the consumer (e.g. Columbus) is constrained to consume data as is produced by a producer (e.g. Yelp and the municipality of Washington D. C.). However, Columbus can circumvent this restriction by first, plugging the inspection results of Washington D.C. restaurants into Dexter as a table, say `inspection(NAME, ADDRESS, VIOLATION)`, and then, filtering popular restaurants (with a Yelp rating of, say $> 4$) that do not have any health code violations using the following query.

```
q(X, Y) :- yelp(X, Y, Z, W)
           & not withViolation(X, Y)
           & Z > 4
withViolation(X, Y) :- inspection(X, Y, Z)
```

**Example 2.** Suppose Joe Black is an entrepreneur who wants to open a Chinese restaurant in Downtown San Francisco. Joe wants to set up his restaurant in a location that does not have other Chinese restaurants in a 2 mile radius. In addition, he wants to set up his restaurant near a parking structure so that his customers can conveniently find parking. The data regarding parking meters owned by the SMFTA in San Francisco is available as a CSV file at `https://data.sfgov.org/Transportation/Parking-meters/28my-4796`. Suppose Joe imports the parking data into a table, say `parking(ADDRESS)`. In order to make an informed decision regarding viable restaurant locations, Joe can query the Yelp data on Chinese restaurants in Downtown San Francisco in conjunction with the SMFTA parking data in Dexter using the datalog query `q` shown below.

```
q(Y) :- parking(Y) & not nearBy(Y)
nearBy(Y) :- parking(Y)
             & yelp(X, Y', Z, "chinese")
             & distance(Y, Y', D) & D < 2
```

In this example, we assume that we have a built-in predicate `distance(X, Y, Z)` which is true if the distance between address `X` and address `Y` is `Z` miles.

**Future Work** Currently Dexter expects end-user queries to be expressed in Datalog. As a next step, we plan to make an end-user's querying experience in Dexter more user-friendly and comprehensible using the following techniques.

1. Supporting *conceptual queries* (Halpin and Morgan 2008) to enable end-users to define concepts by integrating the available data and to pose complex queries over these concepts. In addition, the generation and the integration of concepts could potentially be automated (Maedche and Staab 2001) by leveraging the source-metadata (such as column names, types, integrity constraints and provenance).
2. Supporting *query-by-example* (QBE) (Zloof 1975) in Dexter to enable end-users to query the available data in an user-friendly way.
3. Allowing end-users to define constraints on available data and pinpointing constraint violations that lead to data *inconsistencies*.

## References

Agarwal, S., and Genesereth, M. R. 2013. Extraction and integration of web data by end-users. In He, Q.; Iyengar, A.; Nejdl, W.; Pei, J.; and Rastogi, R., eds., *CIKM*, 2405–2410. ACM.

Agarwal, S.; Mohapatra, A.; and Genesereth, M. 2014. Survey of dexter related tools and techonologies. http://dexter.stanford.edu/semcities/TR-DexterRelatedWork.pdf.

Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American* 5(284):34–43.

Bizer, C.; Heath, T.; and Berners-Lee, T. 2009. Linked data - the story so far. *International Journal on Semantic Web and Information Systems* 5(3):1–22.

Elmagarmid, A. K.; Ipeirotis, P. G.; and Verykios, V. S. 2007. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.* 19(1):1–16.

Facebook. 2014. Facebook api. https://developers.facebook.com/docs/reference/apis/.

Garcia-Molina, H.; Ullman, J. D.; and Widom, J. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

Genesereth, M. 2013. Epilog for javascript. http://logic.stanford.edu/epilog/javascript/epilog.js.

Halpin, T., and Morgan, T. 2008. *Information Modeling and Relational Databases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2 edition.

IBM. 2014. Smarter cities. http://www.ibm.com/smarterplanet/us/en/smarter_cities/.

Kossmann, D. 2000. The state of the art in distributed query processing. *ACM Comput. Surv.* 32(4):422–469.

Maedche, A., and Staab, S. 2001. Ontology learning for the semantic web. *IEEE Intelligent Systems* 16(2):72–79.

Mohapatra, A., and Genesereth, M. R. 2013. Reformulating aggregate queries using views. In Frisch, A. M., and Gregory, P., eds., *SARA*. AAAI.

OpenDataDC. 2014. Washington DC restaurant inspection data. http://www.opendatadc.org/dataset/restaurant-inspection-data.

Simon, P. A.; Leslie, P.; Run, G.; Jin, G. Z.; Reporter, R.; Aguirre, A.; and Fielding, J. E. 2005. Impact of restaurant hygiene grade cards on foodborne-disease hospitalizations in los angeles county. *Journal of Environmental Health* 67(7).

Twitter. 2014. Twitter api. https://dev.twitter.com/docs/api/1.1.

YELP. 2014a. Local inspector value-entry specification (LIVES). http://www.yelp.com/healthscores.

YELP. 2014b. Yelp search api. http://www.yelp.com/developers/documentation/search_api.

Zloof, M. M. 1975. Query-by-example: The invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases*.