**Background:**
Over the past 10-20 years, self-describing semi-structured data formats have become increasingly common as a means of storing and sharing information. The most prevalent example of these data formats today is Javascript Object Notation (JSON), which models objects as a key-value map composed of simple primitives. The format is self-describing because users can entirely deduce the structure of a document from its contents. It is semi-structured because it provides syntactical constraints that valid documents must obey, but does not allow documents to require that keys be present or to require that keys map to a specific type. Any mapping of keys to valid JSON primitives (numbers, strings, Booleans, arrays, or objects) qualifies as a valid JSON document.

In the wake of JSON’s rise to prominence, new database systems which specifically target this kind of data have emerged. These systems—often billed under the NoSQL document-store umbrella—allow users to store and query their JSON documents with very little configuration or upfront investment of resources. Documents are often stored within these systems as single units that include all relevant information. Relationships between objects are encoded by nesting objects within one another (either directly or within an array). For example, consider the representation of a Post object which belongs to a Category. A single Post object may map a key “text” to a string and a key “category” to a nested object. The nested object in turn has the keys “name” and “description.” The embedded object represents a one-to-many relationship: every Post is related to a single Category; Categories with the same attributes may appear duplicated across many different Posts records. Significantly, the nature of the relationship between Posts and Categories cannot be modeled within a NoSQL system. Beyond duplicating entire embedded records or implementing an ID scheme within the application layer, it is impossible for the user to indicate at the database level that multiple Posts are associated with the same logical Category.

These NoSQL systems contrast greatly with the relational database management systems (RDBMSes) that have traditionally been used for data storage and retrieval. Most obviously, RDBMSes carry greater upfront costs because users must predefine the structure of their data before storing it within the system. Beyond this technical imposition, however, RDBMSes also encourage different data models than their NoSQL counterparts. Although it is possible to model the Posts and Categories introduced above by creating a single table containing all the relevant fields (“post_text,” “category_name,” and “category_description”), this schema does not leverage the full power of the relational system. Instead of this redundant approach, database designers encourage normalized schemas which eliminate redundancy and interdependence of data elements. This domain, for instance, would be modeled as two separate tables. The “posts” table would have the fields “post_text” and “category_id.” The “categories” table would have the fields “id,” “category_name,” and “category_description.” The RDBMS would be made aware of the association between the two tables through a foreign key constraint linking the “id” and “category_id” fields.
This normalized representation provides two advantages over its denormalized counterpart. First, this representation reduces the amount of disk space required by the database by eliminating the redundant storage of the category’s name and description. Second, this representation prevents updates from introducing inconsistencies into the data. Whereas in the denormalized scheme a partial update could result in the same “category_name” having several different “category_description”s, this update anomaly cannot occur within the normalized representation because the description exists in exactly one location. In general, normalization eliminates needless redundancy and protects the database from logical inconsistencies.

Despite these benefits of normalization, NoSQL advocates argue that normalization and schema design are too burdensome for the user. By embracing denormalized data structures and eliminating schemas, NoSQL systems allow their users to begin work immediately and to quickly adapt to novel forms of data. Moreover, for some queries, the NoSQL systems may perform better as a result of denormalization because they do not incur the costs of joining tables together (though for many queries—including table scans—denormalization has negative performance implications).

**My Proposal:**
For this project, I will build a database prototype that offers both the flexibility of a NoSQL system along with the space efficiency and data integrity advantages of a RDBMS. The system will allow inserts of JSON documents and will allow users to issue SQL queries using the names and relationships described in the inserted JSON. Internally, the system will represent the documents not as a single relation, but as a set of relations in normal form. The schema for this internal representation will be constructed by analyzing the structure of the data itself.

In an initial analysis phase, the user will bulk insert a set of JSON documents typical of the type of data they wish to store. Data dependencies and relationships between objects will be deduced by a series of machine learning algorithms considering features such as repetitions in nested data structures and the strength of the correlation between attributes on the record. These relationships will then be used to create a normalized schema that suitably represents the inserted documents. As future documents are inserted into the database, the schema will evolve to accommodate patterns not seen in the initial analysis phase. Since the system will rely upon statistical measures to form its schema, the system must also include a mechanism for alerting the user of changes to the internal representation of their data and for allowing manual overrides of the automated process.

While this describes the broad contours of my proposed system, I have many questions that presently remain unexplored. These include: Is dot notation (e.g. “parent.child.child”) sufficiently expressive for SQL queries over the highly nested JSON documents that the system will store? When the system decides to represent JSON nesting by creating a new normalized table, how should primary keys be chosen? When two records contain nearly identical nested objects, how should conflicts be resolved and reported to the user? What kinds of logging structures are necessary to undo changes which result from the machine learning algorithms underlying the system?

**Related work:**
Bahamani et. al. and Yazici and Karakaya have addressed the topic of automatic normalization of database schema in previous papers.\cite{2,3} However, both papers presuppose that the dependencies between data elements and the relationships between those data elements have been given by the user. From these descriptions of the data, they algorithmically transform their initial schema into a normal form. While my project may benefit from their work, my efforts are somewhat tangential to theirs because I am assuming no initial knowledge about the structure of the data.

In a less directly related project, Rahm and Bernstein survey automated methods of matching relational schema that are not identical but represent the same data.\cite{4} Although this is not directly the problem my project aims to solve, my system will be presented with an approximation of the schema matching problem every time a new record is inserted because it will need to decide whether the new JSON document maps to an already existing table or if new tables must be defined.

**Deliverables:**

- A collection of 3-4 real datasets which are suitable for experiments
- A machine learning algorithm which can infer normalized schema from data sets
  - Implemented within the Postgres source code
- Experimental results comparing the performance of selected queries on 1) a Postgres schema closely resembling the initial data format with 2) a Postgres schema generated from my machine learning algorithm
- A paper summarizing this technique and the experimental findings

