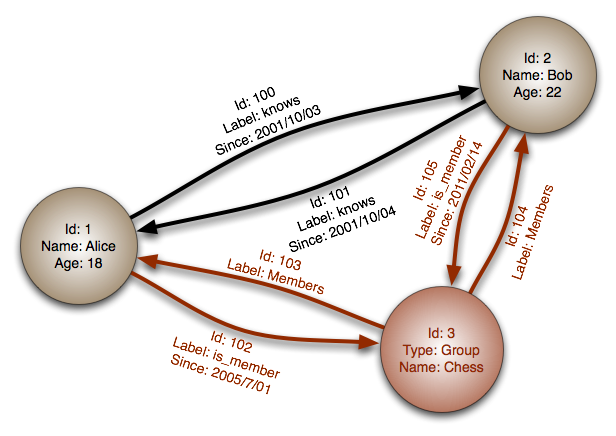
**Implementation of a NoSQL database and interface**

**Background:**

The first non-relational databases were created in the 60s as research interests. Relational databases have dominated the scene for the majority of history, but recent needs of Web 2.0 companies such as Google and Facebook have created a space for NoSQL databases. The defining difference between the two solutions is that NoSQL don’t implement a set of tables connected through a series of relationships. Different types of NoSQL databases use different base data structures in their representation of data. Two common examples are graph based databases and document based databases. In a graph database utilizes a standard graph data model, where the nodes represent entities and the edges represent relationships between the entities. Both nodes and edges can have attributes assigned to them. The relationships defined by edges do not require a common attribute between the nodes. The benefit of using these edges to define relationships is avoiding the heavy cost of joins. It also more intuitively represents non-hierarchical data such as a social network. These types of databases are often used to implement applications that make recommendations, such as the “Recommended for you” feature on Amazon. The diagram below explains how these connections work.



The second type of database, the kind used in my implementation, is a document based database. Here, entities are represented by a document. These documents follow one of the standard formats such as JSON or XML and contain a series of key-value pairs. Each document would be analogous to a tuple in a relational model. The key’s must be Strings but the values can be Strings, numbers, arrays, or even an entire other JSON. An example of one of this documents is below. \_id is used by the database to uniquely identify the document and is also used to access it in constant time through a hash.



Documents are grouped into collections. These collections do not impose any schema on the documents, it just implies a logical connection between the documents. Relationships can be created in documents through two main means. The first is similar to relational database, where an attribute is used to identify another document. In the above example, this is done with location\_id. You can then use a function called aggregate to query the documents based on these attributes. The second, more efficient method is to embed a document inside of another one by including it’s \_id. This is much more performant, because there is no need to search the entire collection, the document can be retrieved directly. Below are two documents. The first is what is actually persisted in the document while the second is the result of retrieving the document by it’s \_id and embedding it.



NoSQL databases choose to implement more programmatically friendly data structures such graphs or documents. This provides a much more natural connection with the program structures that access the database. This can be seen most heavily through the general absence of Object Relational Mappings in NoSQL project. Because the data is stored in the database in a similar structure to how it is stored in the program layers, there is no need for a complex mapping. Simply translate easily through the two formats. For example, I chose to use MongoDB which is a document-based database. Mongo stores a single entity as a JSON document. This allows it to easily couple with loosely typed languages such as Ruby and Javascript. The documents are just parsed into the program layer without any kind of mapping. In addition to easy integration with many programming languages, NoSQL databases are schema-less. This allows for greater flexibility in development which is beneficial for many modern development practices such as Agile methodologies.

**Implementation:**

When deciding on a plan for implantation, I focused on leveraging all of the benefits of NoSQL database in order to fully understand them. I chose to use the MeteorJS framework. This javascript framework allows for full-stack development in javascript, a loosely typed language that integrates heavily with the document based MongoDB. In fact, Meteor is specifically designed to be used with MongoDB. This allows me to write my entire project, including all client and server code, in Javascript. This speeds up the development process and allows for very simple and fast programming. If I want to add a new entity to my project, I can simply create functions to perform basic CRUD operations and create a collection in the database.

There were a few small changes that needed to be made to the relational database schema to migrate it to Mongo. Most of these involve removing relationships, since Mongo does not have a concept or a relationship. Instead, these relationships are modeled as imbedded document ID’s. If Object A has a relationship with Object B, then Object B will have some key in it that represents that relationship, and the value for that key will be the ID of the document. This shows the most in our Sales and Inventory Transfer entities. Specialization is much easier to implement in NoSQL because it is schema less. Instead of one parents table with child tables for the subtypes, you can simply store all the data in one document with a key that marks the type of specialization. These documents can be stored in the same collection. For example, there are two types of employees, staff and manager. Instead of one employee collection, one staff collection and one manager collection, we just store it all in the employees collection. Each employee document has a key called “subtype” that has the value of either manager or staff. Again, we can see the flexibility of NoSQL’s schema-less nature. If we wanted to add a new subtype, all it would require is entering the data. We don’t have to define a whole new table and relationship.

Because NoSQL databases attempt to be more programmatic in their design, they are not searched through queries like in SQL. Instead, it implements functional access to the database. In your program, you can obtain objects that represent an entire collection, and the call collection.find(). This function takes an JSON object as it’s argument, and this JSON defines the search criteria for that. This makes searches easy as well, because you can simply create an object that has the attributes you want and pass that in rather than parsing through the inputted data to create a query. However, the programmatic approach to queries is where the main weakness of NoSQL comes into play. Searching highly relational data is difficult without the ability to create intricate JOIN queries like in SQL. For this project, this challenge is seen in the Sales and Inventory Transfer entities. These are almost exclusively relational entities. If, for example, we wanted to find all of the employees that have been part of a sale, it would require a more complex query. You have to use a special operator to perform an “aggregation” of the documents and then query that aggregation.

Schema-less design does not mean that your data does not have a schema. However, unlike in a traditional relational database, that schema is implicitly defined by the application accessing the database not the database itself. In my project, each of the entities is assigned a schema through the collection2 and simple schema javascript libraries. This defines a format for each collection and is used for data validation in the UI as well as in inserts. An example of one these schemas as well as the code used to apply those schema to different UI elements and database operations is below.

**

Here, we define a schema for the customers collection, and assign its validation context to the insert function used to add a new customer. This ensure that the implicit schema will be upheld, but does not impact the database in any way.

**Differences between NoSQL and SQL implentation:**

As has been highlighted many times in this report, the major difference in developing with MongoDB is the flexibility of the project. You no longer have to have rigidly designed schemas in advance and as the needs of the project change, you can adapt your solution to those changes. Decisions that are made early in the project can be easily changed without major consequences. Relational databases are far less flexible. Add new attribute to a table requires re-defining the schema, modifying your ORM, modifying your data and changing the interface to accept this new attribute. With Meteor and Mongo, you can simply modify your data and the entire program, if written properly, will adapt to this new data. However, as I have explained, it is much more difficult to create complex queries. The structure of calling functions instead of queries requires a very different way of thinking about the data and its structure that is not as easily intuitive as an SQL query.