2017

Extracting a Relational Database Schema from a Document Database

Jared Thomas Wheeler
University of North Florida, n00662180@ospreys.unf.edu

EXTRACTING A RELATIONAL DATABASE SCHEMA FROM A DOCUMENT DATABASE

by

Jared Wheeler

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2017

The thesis "Extracting a Relational Database Schema from a Document Database" submitted by Jared Wheeler in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

_____ _____

Dr. Behrooz Seyed-Abbassi
Thesis Advisor and Committee Chairperson

_____ _____

Dr. Ken Martin

_____ _____

Dr. Roger Eggen

Accepted for the School of Computing:

_____ _____

Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____ _____

Dr. Mark Tumeo
Dean of the College

Accepted for the University:

_____ _____

Dr. John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

I took a long time to reach this point, to complete my thesis. I would like to thank my thesis advisor, Dr. Abassi, for his patience and assistance through all of my problems, as well as his experience and guidance for writing this document. Thanks to my thesis committee, Dr. Eggen and Dr. Martin, for their support and help to drive this to completion. My thanks to Professor Schuller for giving me encouragement when I was at my wit's end. And finally, to my friends, managers, coworkers, and family for their understanding and unending support.

CONTENTS

FIGURES

TABLES

ABSTRACT

As NoSQL databases become increasingly used, more methodologies emerge for migrating from relational databases to NoSQL databases. Meanwhile, there is a lack of methodologies that assist in migration in the opposite direction, from NoSQL to relational. As software is being iterated upon, use cases may change. A system which was originally developed with a NoSQL database may accrue needs which require Atomic, Consistency, Isolation, and Durability (ACID) features that NoSQL systems lack, such as consistency across nodes or consistency across re-used domain objects. Shifting requirements could result in the system being changed to utilize a relational database. While there are some tools available to transfer data between an existing document database and existing relational database, there has been no work for automatically generating the relational database based upon the data already in the NoSQL system. Not taking the existing data into account can lead to inconsistencies during data migration. This thesis describes a methodology to automatically generate a relational database schema from the implicit schema of a document database. This thesis also includes details of how the methodology is implemented, and what could be enhanced in future works.

Chapter 1

BACKGROUND

Relational databases were first introduced in 1970 by E.F. Codd [Codd70]. Relational databases are based on set theory and relational algebra to store related data in a structured manner. These databases consist of tables that are made of columns. Columns specify a data type which can be stored.

In database tables, columns can also have constraints. Constraints are criteria that must be fulfilled for a relational database to accept the data being inserted. Constraints can be that every record has a unique value, or if empty a default value is utilized. A relationship is created through a foreign key constraint, which requires a record with the same value of that column to exist in another table. Information is retrieved from a relational database by querying a table, and performing joins on these relationships to correlate data between the tables.

Relational databases have been built upon as major products for technology giants, such as Oracle, Microsoft, and IBM. New feature sets and improvements are still being made to them as these technology giants push to service more needs. Relational databases are still the most widely used database today due to the reliability that relational database systems are known for.

Relational database systems are Atomic, Consistent, Isolated, and Durable, referred to as "ACID". When a data source is ACID, it means that the transactions possess a specific set of properties [Date04]. The "A" stands for atomicity, which means that any changes to the data can be transactional, and therefore if any part of the transaction fails, the change does not take place.

The "C" is consistency, meaning the data will be valid according to rules in the system. These rules are primarily based on constraints, such as unique constraints and foreign key constraints, among others. Foreign key constraints are particularly important as they define the relationships between tables in a relational database. Without consistency, records could be dependent upon other records which no longer exist.

The "I" stands for isolation. Isolation is the property that transactions executing at the same time have no impact until they complete and commit. The "D" stands for durability. Even if the system suffers a failure, such as crash or power outage, all completed transactions are preserved.

Since the turn of the century, with the rise of the Web 2.0 and Internet of Things, a movement away from relational databases has been emerging [Couchbase16]. The NoSQL Movement encourages the use of data structures other than relational database to store data. NoSQL stands for "Not only SQL" [Parker13], and has led to an environment of Polyglot Data, where multiple data storage technologies are utilized for a company's needs [Young2013].

NoSQL represents a change in priorities for databases. The NoSQL databases no longer support all of ACID, but instead prefer responsiveness and flexibility. There are many varieties of NoSQL databases; however, this thesis will focus on the document database.

Document databases store unstructured data in a structure similar to JavaScript Object Notation (JSON), providing flexibility of schema [Couchbase16]. JSON consists of a series of key-value pairs. The values can be strings, numbers, arrays, or further JSON. In Figure 1, the Address field contains a sub-object in JSON. The sample JSON also has the field PhoneNumbers which holds an array of strings.

```
{
    Name: 'John Doe',
    ContactInformation: {
        PhoneNumber: '904-555-8648',
        Address: {
            StreetNumber: '153',
            StreetName: 'Pine Lane'
        },
    }
}
```

Figure 1 Sample Data in JSON

Different document databases have minor variations upon the foundation of the JavaScript Object Notation (JSON) structure. For instance, MongoDB makes use of a format called Binary JSON (BSON), a compressed version of JSON for more efficient storage [MongoDB16B].

Document databases abandoned ACID in favor of BASE. BASE stands for Basic Availability, Soft-state, and Eventual consistency [Sasaki15]. Basic availability indicates that the system will appear to be online most of the time. Soft-state has less constraining rules around consistency. Soft-state will allow data types of fields to change over time. Also, soft-state means that replicated objects can be temporarily inconsistent. Eventual consistency means that these inconsistencies will gradually be corrected in the long term. Therefore, the system still requires a way to make data synchronize over time, but it has a more flexible time period to do so than an ACID system would.

Following the BASE principals leads to a system with faster reads and writes under large loads because it has fewer constraints trying to protect the data. The BASE principles do, however, result in the loss of isolation, atomic transactions, and referential integrity [Boicea12].

Many document databases target heterogeneous distributed systems [Mohan13]. BASE makes distributed systems easier to utilize due to the eventual consistency, which indicates that all data across the cluster will eventually agree, although there might be some inconsistencies at specific points in time. These systems place a higher importance upon being relatively correct, but accessible, data as opposed to causing slow-downs in reads to ensure complete correctness.

The idea of the document databases embraces this change in philosophy, especially when it comes to utilizing the flexibility of JSON. Soft-state from BASE means that the system

will not constrain the data of the system. Instead, any structure can be saved in a document, as long as it does not violate any of the few requirements the system does have, such as maximum document size. Therefore, documents within a collection could potentially have a wide variety of field compositions. It is left to the application to handle the structure, null values, and potential differences in data type.

Document databases store JSON-like structures. JSON is quite literally a serialization format of objects. Document databases are storing serialized versions of objects as they could appear in the code of an application. This storage is similar to what Object Relational Mappers (ORMs) do for applications on relational databases. Popular ORMs include Microsoft's Entity Framework [Microsoft16] and Hibernate [RedHat16] for Java.

Object relational mappers take an input of a database schema and generate code that facilitates storing the data in that format. The classes generated by the ORMs are usually referred to as Data Transfer Objects (DTOs). The relational database forces the code to adapt to the database, and store in a format that the database will accept.

Document databases can be viewed as the opposite of a relational database when it comes to storing from code. The actual schema within a document database is not enforced by the system, but rather is adaptable to the data stored within it. Therefore, the schema is actually determined by the application that is utilizing it. More accurately, by the DTOs that the application uses to save to and extract data from the storage system. When a

schema is driven by the application, rather than the database, it is called an “implicit schema” [Gomez16].

The main difference between DTOs in an ORM and in a document database is how the storage handles the links between the objects. ORMs typically generate their classes as a class per table, then the classes hold references that account for the relationships within the database. Document databases, on the other hand, will store the related object(s) directly in the document being saved. These structures are referred to as embedded documents or a subdocument.

The actual structure of a document database is a group of collections. These collections store the previously referred to documents, similar to how tables store records [MongoDB16D]. Documents can have subdocuments, which can have further subdocuments, nesting onward as many levels deep as the application requires. MongoDB supports up to 100 levels [MongoDB16C]. In addition to subdocuments, documents can store fields of a variety of data types, and arrays of fields or subdocuments.

Document databases do not support references between documents, and has no concept of relational operations, such as joins. Instead, all data related to a document should be nested within the structure of that document. Therefore, it can become necessary for the same data to be inserted in multiple documents when it is shared. There are scenarios where if the data is updated in one document, it would need to be updated in all

documents that store the same information. Repetitively storing the same data through a database leads to inefficiencies in storage and potential update anomalies, which may result in denormalized data.

This was the scenario that a social media application called Diaspora encountered [Mei13B]. Diaspora was built on MongoDB, a popular document database. After reaching production, the creators encountered problems with subdocuments becoming inconsistent, when consistency was required. After several attempted solutions failed and new features required better consistency, the app was converted from utilizing MongoDB to use a MySQL relational database. The project manager noted the process by which their data was converted to be highly manual and problematic, especially in scenarios with a more denormalized structure [Mei13A].

There is no formalized method for transforming data from a document database into a relational schema [Goyal16]. Data migration from a document database into a relational database is currently handled by one of two methodologies, as described below.

The first method is to flatten the data and import it directly into the relational database. There are existing tools for document databases to export the contained data in a Comma Separated Value (CSV) file, which can then be imported into many modern relational database systems. Method one is a direct import from the CSV format into a single table.

Once the data is imported into the relational system, it can become difficult for the user to transform into the final data structure. The intermediary data structure may be sparse, in other words, full of empty columns. The sparsity of data is due to the ability of the document database to change its structure, leaving fields empty.

Also, it is likely for data to be duplicated across records. Because document databases cannot refer to data between documents, the data is copied or inserted new. These problems lead to difficulties querying data due to potentially large numbers of equivalent columns which must be included in filters or projections separately.

The second method is to create an intermediary format which the data from the NoSQL source could reside in as it is transformed from the original format into that of the final database. The moving of data can be done through custom tools built to transform the data for a specific pair of databases. However, the development process such a custom tool can be costly for the organization funding the work.

Alternatively, more general solutions can be found, such as Microsoft's SQL Server Integration Services (SSIS). Despite using the general tool, the user must still have some technical knowledge to be able to architect the tables in the relational database. If the migration process is done incorrectly, data could be improperly associated or the relational system could suffer sparse and duplicate data, like in method one.

The difficulty of migrating the schema of the data from the document database source to a relational database target reveals that an expert in the data is required to determine an appropriate schema to be able to accurately migrate from a document database to a relational system. That expert must be proficient in database design to avoid the dangers of duplicate or sparse data, as neither methodology accounts for the design and creation of the destination database. The expert must also have strong knowledge of the domain that the system operates in, to be able to create the appropriate relationships of the target database.

A better methodology is needed that would automatically derive schema from the data of a document database to eliminate much of the need for an expert to create the database schema. Once the new schema is created, existing tools with friendlier interfaces could be used to migrate the data from the document database to the relational system. No tools or proposals exist today that perform automatic schema generation based on a document database.

A better methodology is needed that would automatically generate a relational database structure from a document database to ease the transition from document database to relational database. Document databases are popular, and some may argue overused [Dziruko15]. Relational database systems still have advantages when data is highly related or cases when ACID transactions are required. Relational databases are also more developed and integrated with common business software for reporting and analysis. If a

project wrongly chose to utilize a document database, the path back to other data models can be a painful one, as seen in the previously referred to Diaspora scenario [Mei13A].

In chapter two, more core ideas will be explained. Chapter three will feature an explanation of the new methodology by which a relational schema can be automatically extracted from a document database. Chapter four contains details about the implementation of such a system. Chapter five will offer an analysis of the results from running the system on a set of sample databases, and chapter six will offer conclusions and directions for future works.

Chapter 2

CORE CONCEPTS

The foundation of a relational database is the table. A table is made of a collection of columns and rows. Each column stores data in a predefined format. Each row, also referred to as record or tuple, in the table represents a set of data in the format defined by the columns. A primary key constraint can be defined on a column, or group of columns, to uniquely identify each row. A primary key is a column, or collection of columns, that is guaranteed to be unique and not null across all rows [W3Schools17]. A table can only have one primary key.

While rows within a single table are independent of one another, rows can be associated across different tables. The association of rows across multiple tables is referred to as a relationship. Tables can be related to one another through foreign key constraints. A foreign key consists of two columns, in separate tables, which are associated by the constraint. The constraint is associated directly to one of the tables. For the specified column in that table, referred to as the foreign key, to contain a value, that value must first exist in the specified column of the referenced table. This forces consistency between the tables related in the foreign key relationship. A table can have multiple foreign key constraints.

2.1 Relationships

The ability to relate data through foreign keys allows the database design to distribute data across multiple tables. The term "schema" is usually used to refer to the overall structure of the columns, tables, and relationships.

Cardinality of the relationship identifies how information within each table relates to information of other tables. Relationships can be described as one-to-one, one-to-many, or many-to-many, referring to how the data between tables can be related. The definitions in sections 2.1.1 through 2.1.3 describe binary relationships, those consisting of two tables. Definition 2.1.4 will address relationships consisting of a size other than two tables.

2.1.1 One-to-One Relationship

A one-to-one relationship specifies that for the relationship between two tables, there will only exist one row in each that will be related to one another. There are multiple ways to facilitate the one-to-one relationship in a database schema. Figure 2 features three one-to-one relationships, referred to as A, B, and C. A true one-to-one relationship is demonstrated in A. The two tables share a primary key, therefore guaranteeing that only one record on each table will be related. The one-to-one guarantee is due to the unique requirement of primary keys, therefore the key can never be repeated in either table. The relationship between TableA and TableB is known as an identifying relationship, because

the primary key in TableA is also part of the primary key in TableB. The identifying relationship indicates that a record in TableB will never exist without a corresponding record in TableA.

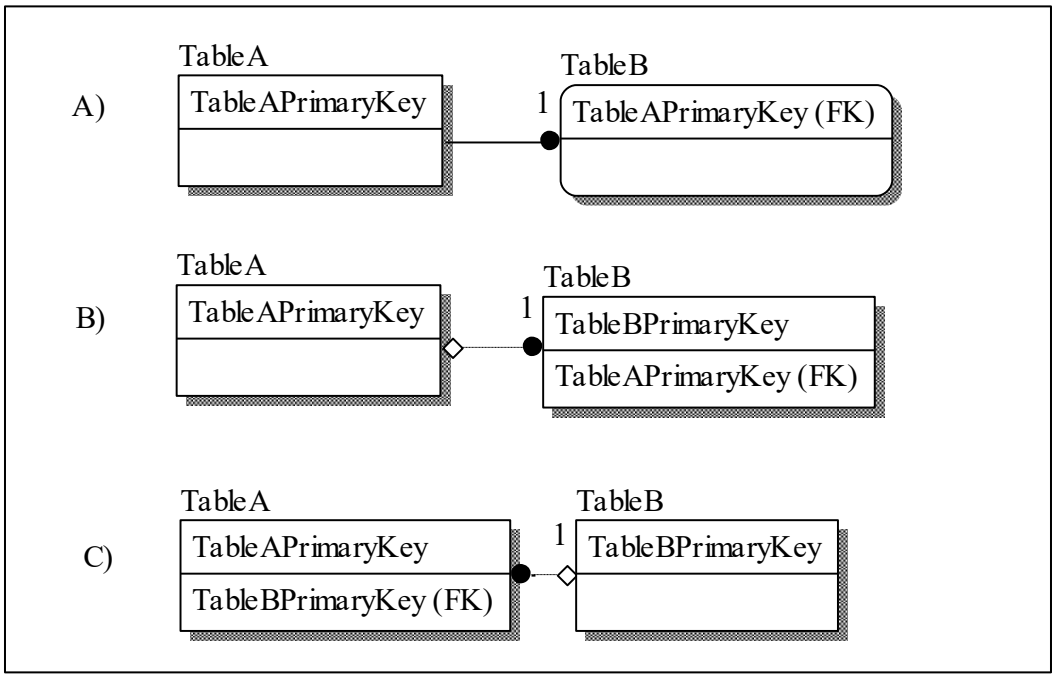

Figure 2 One-to-One Relationships

The relationships in B and C are logically equivalent to relationship A, though less restrictive. The TableB in example B has the foreign key to TableA in a normal foreign key field, with no unique constraint. Therefore, it is physically possible for multiple records in TableB to relate to TableA; however logically it can be utilized as a one-to-one relationship. Example C is the same as relationship B, only the keys have been reversed. The key for Table. Relationships B and C physically facilitate a one-to-many relationship.

2.1.2 One-to-Many Relationship

A one-to-many relationship allows for many rows in one table to be related to a single row in another table. In Figure 3, TableA and TableB have a one-to-many relationship. If TableB is filtered by the TableAPrimaryKey, then multiple records could be returned. Alternatively, if the same filter was applied to TableA, then only one record would be returned. Because TableB is associated to one record of TableA, but that same record in TableA is associated to multiple records of TableB, this is a one-to-many relationship.

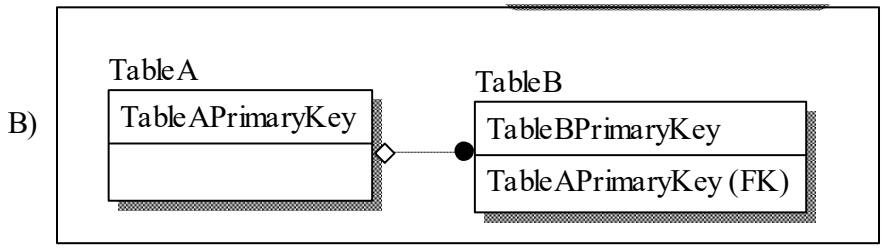

Figure 3 One-to-Many Non-Identifying Relationship

Another variation of the one-to-many relationship is the identifying one-to-many relationship, shown in Figure 4. The difference between an identifying and a non-identifying relationship is the primary key of TableB, in the example. In the identifying relationship, the TableAPrimaryKey foreign key is above the solid line, indicating that it is part of the composite primary key. TableBPrimaryKey and TableAPrimaryKey are combined to serve as the primary key. TableAPrimaryKey still acts as a foreign key referring to TableA from TableB, but it also means that the data in TableB will not exist without the related data in TableA.

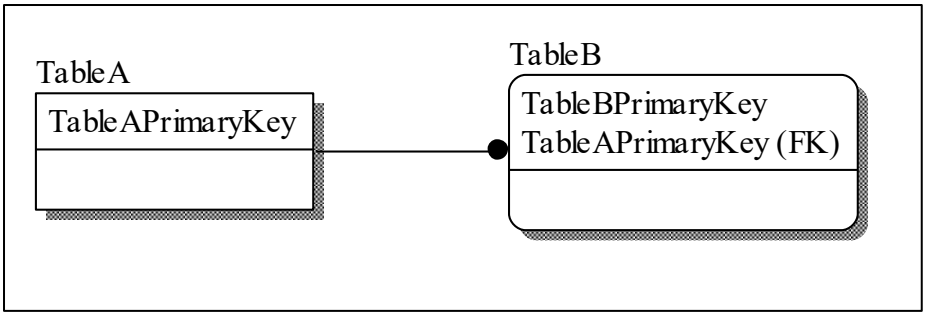

Figure 4 One-to-Many Identifying Relationship

Unlike the one-to-one identifying relationship in Figure 2 A, the one-to-many identifying relationship requires TableB to have a composite key. Primary keys must uniquely identify a row. With only the TableAPrimaryKey as a primary key of TableB, there could only be one record in TableB with that value, which is the one-to-one relationship. The composite key allows multiple rows to exist for each value of the TableAPrimaryKey value, therefore creating the one-to-many relationship.

2.1.3 Many-to-Many Relationship

A many-to-many relationship allows many rows from each table to be related to one another. Figure 5 shows the logical diagram of a many-to-many relationship. When the data of the two tables, TableA and TableB, are joined together, a user would be able to find multiple records of data in TableB using the TableAPrimaryKey as a filter. The user would also be able to find multiple records of data in TableA using the TableBPrimaryKey. In the one-to-one and one-to-many relationships, the filtering of data was possible by one table holding the primary key of the other table in a foreign key; however a many-to-many relationship is handled differently.

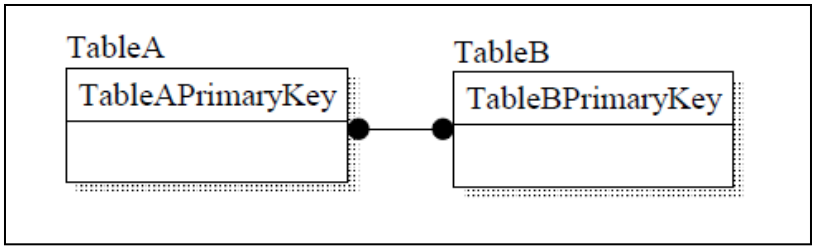

Figure 5 Logical Many-to-Many Relationship

In standard relational databases, many-to-many relationships are not directly supported. Instead, a bridge table is created, which consists of two one-to-many relationships. The BridgeTable stores the primary keys from each of the tables in the many-to-many relationship. When the three tables are joined together, it gives the appearance of the many-to-many relationship. In Figure 6, if TableA or TableB were filtered based on the combinations within the BridgeTable, many records could be returned from each table. The possibility of many being returned from each makes it a many-to-many relationship.

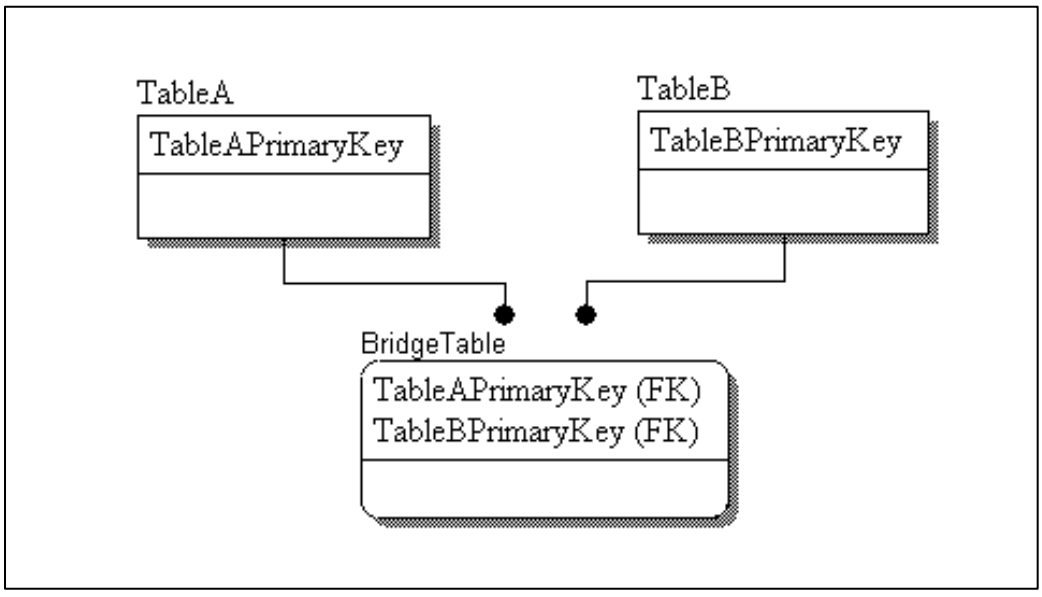

Figure 6 Many-to-Many Relationship

2.1.4 N-ary Relationships

As the many-to-many relationship alludes to, while relational databases only support key constraints between two tables, there can be more complicated logical relationships. Logical relationships can be referred to as unary, binary, ternary, and up, or summed up as n-ary.

A unary relationship is when only one table is involved. The table has a foreign key which refers to is own primary key. Figure 7 shows how this relationship is built. TableA has the TableAForeignKey which refers back to the TableAPrimaryKey. Unary relationships are often used for hierarchical structures stored within a single table, such as employees with a relationship to their manager.

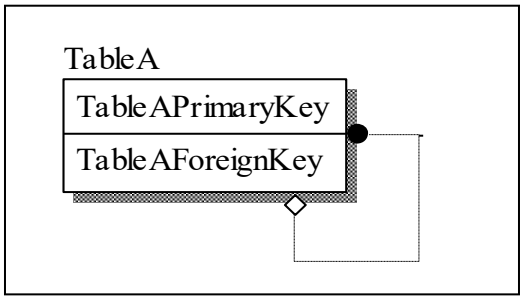

Figure 7 Unary Relationship

N-ary relationships start at the ternary relationship, which relates three tables. The relationship size then scales upward. The n-ary relationships are handled in the same way as a many-to-many relationship, with a bridge table. Figure 8 displays a ternary

relationship. Like the many-to-many, the bridge table holds a key relating to each of the other tables.

N-ary relationships are the foundation for the star and snowflake schema patterns used in data warehousing. These schema feature a central bridge table, referred to as a fact table, which links many other tables with details [Date04].

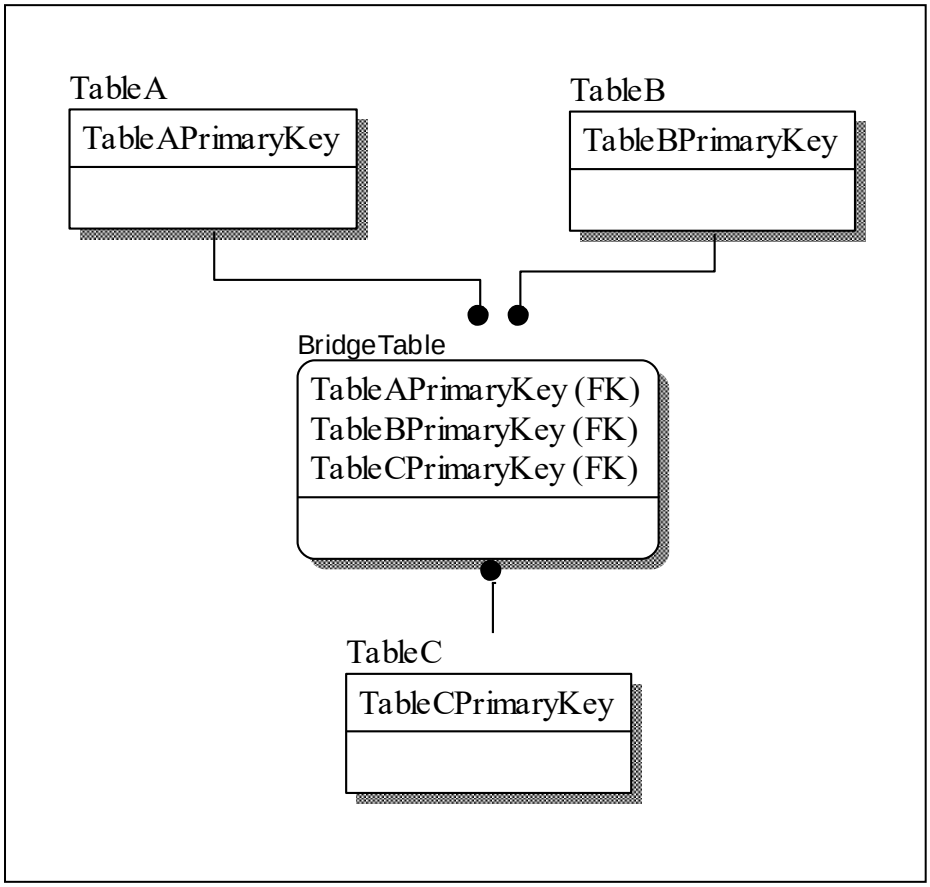

Figure 8 Ternary Relationship

2.2 Normalization

With the ability to separate data across multiple tables, there comes a question of how to design the structure of the data. Normalization is the act of decomposing the data into tables that results in a more desirable structure with less duplication. Normalization is usually described by various normal forms. When the structure of a database satisfies certain sets of conditions, then it can be defined as belonging to a particular normal form. Common normal forms include first normal form, second normal form, third normal form, and Boyce/Codd normal form (BCNF) [Date04].

As a database schema becomes more normalized, the redundancies in data are decomposed into other related tables. Normalization is typically used to spread data out for smaller writes to the database and reduce reading overhead when only a portion of data is necessary to be returned. When the tables are normalized, they could be denormalized again by joining the decomposed tables back together. Denormalized schema is faster to retrieve large sets of data, although can take up more storage space due to redundancies in data.

A sample of a denormalized schema in Figure 9, and a more normalized version of the same schema is in Figure 10. The fields in the denormalized schema beginning with "Address" were extracted into a one-to-many related table, reducing the complexity of the schema and increasing its flexibility. The denormalized schema would encounter

empty fields when a client only had one address. The normalized schema can have one or more addresses without any unintentional empty fields.

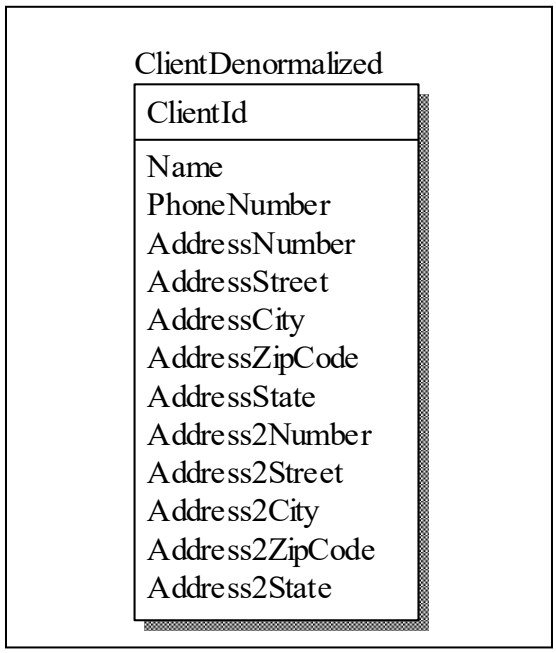

Figure 9 Denormalized Client Schema

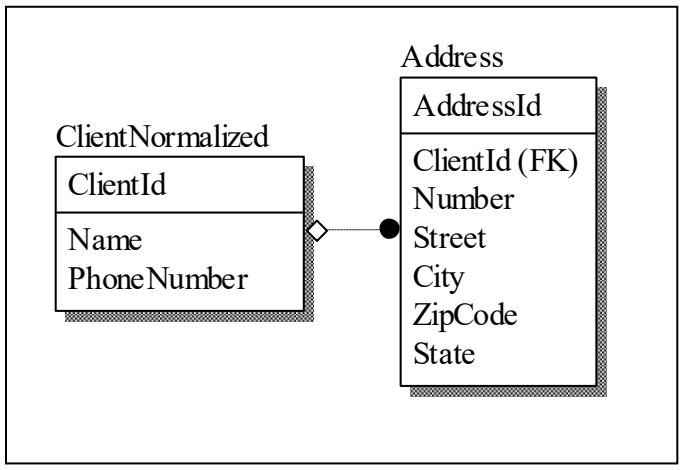

Figure 10 Normalized Client Schema

2.3 Structured Query Language

The Structured Querying Language (SQL) is the standard querying language for relational databases as defined by the International Organization for Standardization (ISO) and American National Standards Institute (ANSI) [Quality Nonsense17]. SQL consists of Data Definition Language (DDL) and Data Manipulation Language (DML).

DDL contains operations such as create and alter table, which specify the structure of the data in the database. DML contains operations such as insert, update, delete, and select, which manipulate the data within the structure of the database. With the select statement, users are able to retrieve data. An important option within the select statement is the join operation, which allows the query to gather data from multiple tables.

2.4 Document Databases

The document database is a NoSQL alternative to the traditional relational database. For the reasons covered in chapter one, document databases are designed to be highly available caches of data. Document databases do not consist of tables, but rather collections.

Document databases are built of collections which hold root-level documents. These documents can hold fields, lists, and subdocuments. Subdocuments are additional

documents embedded within another document. Lists can consist of simple values or subdocuments [Gyorodi15].

The storage medium of choice for document databases is JSON, or JSON-like structures. JSON is a popular data-interchange format based on a subset of JavaScript [JSON16]. JSON is composed of key-value pairs which can store simple values, arrays, or further JSON objects.

JSON boasts a simplicity that other options, such as eXtensible Markup Language (XML), lack. While XML requires opening and closing tags for every element, JSON focuses on key-value pairs within curly braces. The lack of closing tags cuts back on the amount of data required to define values, especially of multi-layer structures.

Table 1 demonstrates the difference in length for the serialization of a small object which is composed of two properties, the second of which has multiple layers to it. In the example, the object serialized in XML is 148 characters, while the same object serialized in JSON is 83 characters. For these small sizes, the difference is minor, however, the scenario amounts to a 44% length reduction. If scaled to a larger object, the size difference would be similar.

JSON	XML
{object:{firstProperty:'value',secondProperty:{subProperty:{deepProperty:'value'}}}	<object><firstProperty>value</firstProperty><secondProperty><subProperty><deepProperty>value</deepProperty></subProperty></secondProperty> </object>

Table 1 JSON vs XML

2.5 Variety

Chapter one referred to the implicit schema within a document database, created not by the database, but rather by the application that uses the database [Gomez16]. There have been previous attempts to explore the implicit schema. Most notable is an open source tool called Variety that was published to GitHub in 2012 [Dvorak16].

Variety is designed to examine a MongoDB data source and output metadata about the targeted collection. More specifically, Variety outputs information about the structure of the documents, the data types of the fields, and how frequently each field is present. How Variety determines the schema of the MongoDB will be expanded upon in chapter four. Variety is implemented in a JavaScript framework called Node, which will be covered in section 2.7.

2.6 MongoDB

MongoDB is one of the most popular document databases available to users. Rather than relying on JSON for storage, MongoDB utilizes Binary JSON (BSON), which is an extension of JSON [MongoDB16B]. BSON provides a more efficient storage mechanism, as well as expands on the data types that can be supported.

While document databases have abandoned ACID in favor of BASE, as discussed in chapter one, MongoDB has not left it entirely behind. MongoDB does support a limited form of transactions, where changes to a single document can be transactional when using a keyword [MongoDB16A]. MongoDB recommends a two-phase commit pattern be implemented within the application performing the actions if transactions are needed for anything beyond a single document.

2.7 Node

Node is a server-side JavaScript framework [Node16]. Node boasts a highly concurrent language through lack of locking. Node works because it runs on Google's V8 JavaScript engine in a server-side environment, allowing Node programs to be run without a browser. This has made Node popular for small server-side tools.

Node is frequently used in conjunction with Express web server, AngularJS client-side, and MongoDB database. The combination of MongoDB, Express, AngularJS, and Node

is referred to as a "MEAN stack" [Leanos16]. Due to the common JavaScript functionalities of these technologies, the MEAN stack makes it simple to pass JSON objects between the layers. The MEAN stack is therefore favored by developers who are comfortable with JavaScript and wish to simplify the transferring of data between layers.

2.8 Entity Framework

Entity Framework, as previously mentioned, is an ORM for .NET. Entity Framework has three different modes: code-first, model-first, and database-first. Depending on the mode, Entity Framework will take a different set of inputs. By the end of the process, there will be a database set up with DTOs ready for the application to use.

For code-first, Entity Framework takes a series of DTOs as input. Therefore, the DTOs are created by the developer. Entity Framework will analyze the DTOs and determine an appropriate database schema. The database schema is then inserted into a specified database to prepare it for use with the application.

Entity Framework's model-first database generation allows the user to use a database designer to visually create a database schema. Entity Framework will then generate both the database and DTOs for the application to use.

This thesis will utilize database-first Entity Framework during the methodology evaluation. In database-first, the user provides Entity Framework with a pre-existing

database. The developer must then specify what tables in the database require DTOs. Entity Framework then generates the necessary DTOs to represent those tables and their relationships.

Chapter 3

METHODOLOGY

What makes the described methodology unique is the automatic schema generation. Tools exist which can move the data into a work table or pre-existing schema for Extraction, Transformation, and Loading (ETL) activities, but none of these tools automatically determine an appropriate schema to generate. The methodology described in this chapter derives a relational schema from the structure of a document database. The methodology assumes that the user is already able to extract or determine the implicit schema of the document. Determining the implicit schema may be done by using a tool such as Variety [Dvorak16], which will be described in section 4.1.

The problem that the methodology must overcome is determining the items that document databases do not explicitly track, namely relationships and data types. Determining the relationships between a document and its subdocuments, especially normalizing these relationships, is the more pressing problem. Without normalization, the final schema will facilitate duplicate data and unnecessary null fields, leading to wasted space in the database. Relationships between documents and subdocuments must be discovered and extracted into a relational table structure. Document databases do not have specified data types, therefore a strategy for associating a data type for each column must be created.

The methodology follows the idea that each document can be extracted into a relational table. Tables are generated from each field that contains a subdocument, therefore the name of that table will be the same as the field which stored the subdocument. This is the same for arrays of subdocuments. The name of the root table should be derived from the name of the collection in the document database. Therefore, when a document has a subdocument, the document and the subdocument become two tables with a relationship between them. The extraction of this relationship will lessen the sparsity of columns and decrease the amount of duplicate data.

To properly normalize the relationship between a document and its subdocument, the cardinality of the relationship must be determined. Once that has been determined, an appropriate relationship can be created within the database. The following defines how the methodology determines what kind of cardinality exists, and therefore what kind of schema should be generated.

In general, the relationship chosen depends on two factors: arrays and duplication. Document databases support documents having an array of subdocument. When an array exists, it indicates that multiple of the subdocument can be associated to a single of the parent document. Also, duplication of the subdocument will impact the relationship chosen. If an identical subdocument is present for multiple parent documents, then many parent documents are associated with that single subdocument. From these two pieces of knowledge, the cardinality of the relationship can be determined. From the cardinality, the appropriate schema relationship will be generated.

A series of steps will be repeated for all generated tables. When the schema for a new table is created, a primary key will also be generated. This will be an auto-incrementing integer for simplicity. All fields that are not subdocuments will be directly converted to columns on the table that represents that document.

3.1 One-to-One Relationship Transformation

A one-to-one relationship is found in a document database when a document contains a field which is a single subdocument, and that field is not duplicated in any other documents. When the subdocument matches these criterion, it can be extracted into a one-to-one relationship for the generated schema.

The table derived from the document will have the primary key of the table derived from the subdocument added to it. The key originating from the subdocument will have a foreign key constraint referring to the table derived from the subdocument. This relationship constraint allows for the One-to-One relationship within the relational database.

This methodology for facilitating the one-to-one relationship is less constrained than if the two tables share a primary key, such as example A in Figure 2. However, a shared primary key implies a strong semantic relationship between the tables. Due to the generality of the methodology to be applied in many circumstances, it would be unwise to assume that this strong semantic relationship will always exist in one-to-one

relationships. Therefore, the author chose to err on the side of caution to avoid any such implications in the generated schema.

The transformation is demonstrated in Figure 11. Note that the field SubDocument is a single subdocument. Also, within the entire collection, it is not duplicated. These characteristics indicate the presence of a one-to-one relationship. The schema on the right links the documents such that each DocumentCollection will have a single SubDocument. The columns are made from the other fields on its respective document.

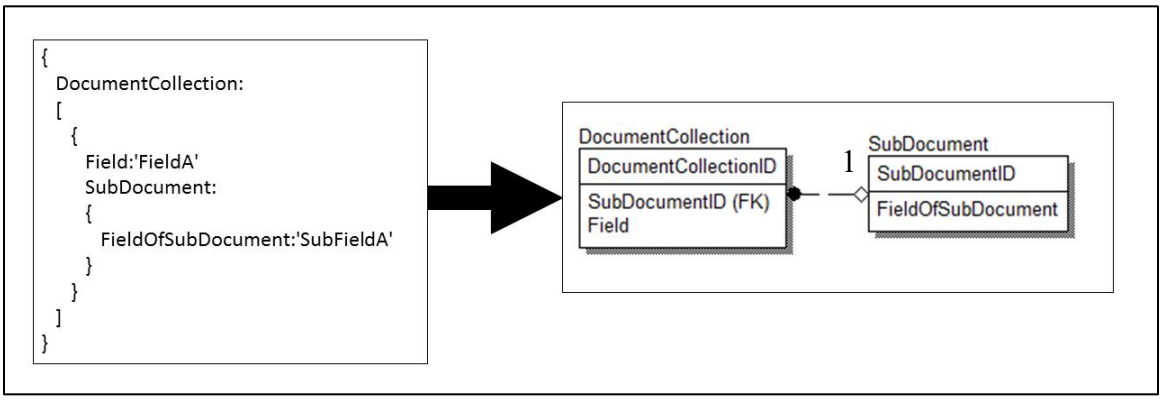

Figure 11 One-to-One Relationship Transformation

Reviewing the sample in Figure 11, the source document contains a field called Field; this became the column Field on the DocumentCollection table. The source document also contains a subdocument in the field called SubDocument. The second table in the generated schema is named SubDocument, for the field that the subdocument was found in. The key that was generated for this relationship between the document and subdocument is called SubDocumentID, also named after the field which held the subdocument. The key is also found on the DocumentCollection table to facilitate the

relationship between the tables. In the source document, the SubDocument field contains a field called FieldOfSubDocument; the field was transferred onto the SubDocument table in the generated schema. By joining these two tables, DocumentCollection and Subdocument, all of the data held in the original document would be retrieved.

3.2 One-to-Many and Many-to-One Transformations

In section 2.1.2, the one-to-many relationship was discussed as a relationship of one row in TableA correlating to multiple rows in TableB. When applying this idea to a document database, it becomes clear that this description of a one-to-many relationship is incomplete.

In a relational database, there can exist a one-to-many relation from TableA to TableB, there can, alternatively, be a one-to-many relationship from TableB to TableA. In a relational database, both of these can be referred to as a one-to-many relationship, because direction is arbitrary.

A document, meanwhile, always starts at the top and moves into deeply nested layers. Therefore, order of the tables cannot be arbitrarily changed. A one-to-many relationship of TableA to TableB would be different from a one-to-many relationship of TableB to TableA. The second relationship can be described as a many-to-one relationship, which becomes necessary when describing the transformation of a document into a relational schema. Section 3.2.1 will discuss detection and handling of a one-to-many relationship.

Section 3.2.2 will expand on the many-to-one relationship, and how it differs from a one-to-many in detection and handling. Section 3.2.3 dives into the difficulties surrounding identifying relationships within one-to-many and many-to-one relationships.

3.2.1 One-to-Many Relationship Transformation

A one-to-many relationship is similar to the one-to-one relationship previously discussed. The difference is that the field within the document is actually an array of multiple subdocuments. However, once more, all objects within the array must be unique to the document.

The table derived from the field will receive the primary key of the document table as an additional column. A foreign key constraint will be placed on the field table to refer to the document table. Figure 12 demonstrates a one-to-many relationship. Note that the SubDocument field is actually an array of subdocuments. The array indicates that there are multiple SubDocuments that will be related to a single parent document. However, there are no duplicates of the subdocument across the collection. Therefore, this example is a one-to-many relationship. With the generated schema, multiple SubDocument records can be associate to a single DocumentCollection record.

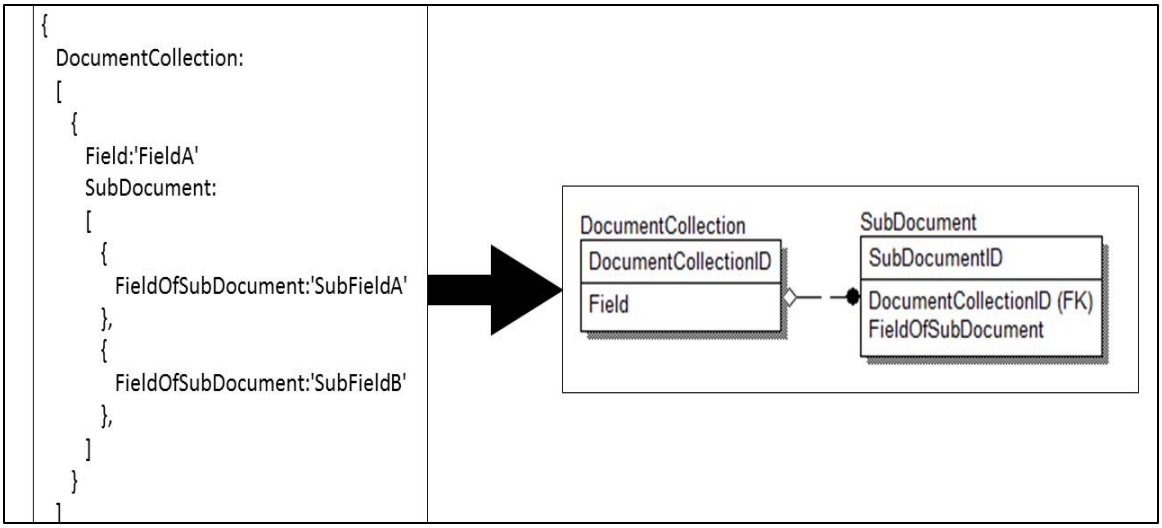

Figure 12 One-to-Many Non-Identifying Relationship Transformation

Reviewing the sample in Figure 12, the source document contains a field called Field; this became on the column Field on the DocumentCollection table. The source document also contains a subdocument in the field called SubDocument. The second table in the generated schema is named SubDocument, for the field that the subdocument was found in. The key that was generated for this relationship between the document and subdocument is called SubDocumentID, also named after the field which held the subdocument. The key is also found on the SubDocument table to facilitate the relationship between the tables. In the source document, the SubDocument field contains a field called FieldOfSubDocument; the field was transferred onto the SubDocument table in the generated schema. By joining these two tables, DocumentCollection and Subdocument, all of the data held in the original document would be retrieved.

3.2.2 Many-to-One Relationship Transformation

A many-to-one relationship exists when an identical subdocument is contained in the same field across multiple documents. Unlike the one-to-many relationship, no arrays are involved. In the many-to-one relationship, one subdocument is related to many instances of the parent document. To generate the appropriate schema, the primary key of the table derived from the subdocument will be added to the table derived from the document. The key that is present in both tables can then be related with a foreign key constraint.

In Figure 13, both sample documents contain an identical subdocument, which is not part of an array. For the purpose of simplicity, the methodology searches for exact matches, so all present fields must have the exact same value. The sameness applies to all subdocuments as well. So, the subdocument must be completely identical through all layers of subdocuments. If there exists a subdocument that is completely identical across multiple documents, then it is considered to be a duplicate.

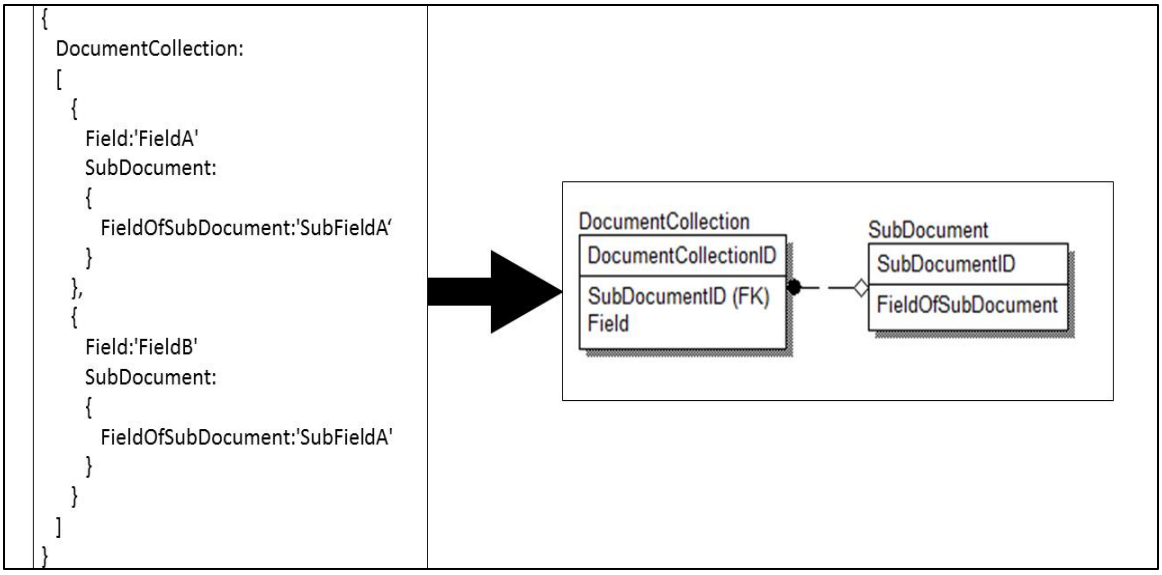

Figure 13 Many-to-One Non-Identifying Relationship Transformation

Reviewing the sample in Figure 13, the source document contains a field called Field; this became on the column Field on the DocumentCollection table. The source document also contains a subdocument in the field called SubDocument. The second table in the generated schema is named SubDocument, for the field that the subdocument was found in. The key that was generated for this relationship between the document and subdocument is called SubDocumentID, also named after the field which held the subdocument. The key is also found on the DocumentCollection table to facilitate the relationship between the tables. In the source document, the SubDocument field contains a field called FieldOfSubDocument; the field was transferred onto the SubDocument table in the generated schema. By joining these two tables, DocumentCollection and Subdocument, all of the data held in the original document would be retrieved.

3.2.3 One-to-Many Identifying Relationships

As with the one-to-one relationship, the one-to-many and many-to-one relationships can also be identifying in a relational database. A relationship is identifying when one table's primary key contains the primary key of the other table in a relationship. Figure 14 shows an example of an identifying one-to-many relationship. TableB has a composite primary key of TableAPrimaryKey and TableBPrimaryKey. TableAPrimaryKey is also used in a foreign key relationship to TableA.

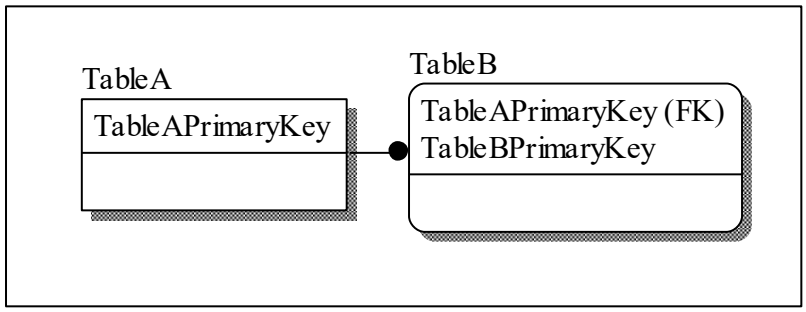

Figure 14 One-to-Many Identifying Relationship

Detecting the special scenario of an identifying relationship within a document database is a difficult scenario. An identifying relationship can be identified in a database by the composite key. Documents may or may not have identifiers in the subdocuments acting as links to the parent document. Dynamically determining that a field is both an identifier and referring to the parent document becomes difficult, as naming could vary vastly. An additional requirement of an identifying relationship is that a record in the child table, TableB in the example, cannot exist without the parent.

Describing a data point as not being capable of existing without another is a powerful semantic statement. Even if it could be determined that a subdocument contains a reference to its parent document, there are still explanations other than an identifying relationship. For instance, the identifier could have existed on the objects to retrieve each other from a different data source, given that either object could be used as an input. It may be possible that a different document collection features the same document as a parent. There are many scenarios which would cause the identifier to be present on the child document without it being an identifying relationship.

To determine the existence of an identifying relationship requires at least two steps. First, there must be a reliable way to determine a field to be referring to the parent document. Second, it must be proven that the object that the subdocument represents cannot exist without the parent document. The second of these steps requires greater knowledge than what can be stored in a document database. Therefore, the methodology treats all relationships as non-identifying relationships.

3.3 Many-to-Many Relationship Transformation

The final relationship that must be transformed from document database to the relational database is the many-to-many relationship. The many-to-many relationship requires the use of a bridge table to accurately represent the logical relationship. The bridge table exists to represent the relationships possible between the two tables that have the many-

to-many relationship. The bridge table may have further information that is unique to each relationship; the methodology will not be taking advantage of that possibility.

A many-to-many relationship occurs in a document database when the field of a document holds an array of a subdocuments, and at least one of those subdocuments is duplicated across multiple documents. Similar to the many-to-one relationship, duplication is judged by a subdocument having identical fields to another subdocument. The difference in a many-to-many relationship is that this duplicated subdocument exists within an array of subdocuments.

The collection of documents in Figure 15 depicts a subdocument that is duplicated across multiple documents. The field SubDocument holds an array of subdocuments. Both documents depicted in the example have a subdocument with the field FieldOfSubdocument that has the value of 'SubFieldA'. This common subdocument indicates the duplication.

```
{
    DocumentCollection:
    [
        {
            Field: 'Field A',
            SubDocument:
            {
                FieldOfSubDocument:'SubFieldA'
            }
        },
        {
            Field: 'Field B',
            SubDocument:
            {
                FieldOfSubDocument:'SubFieldB'
            }
        },
        {
            Field: 'Field C',
            SubDocument:
            {
                FieldOfSubDocument:'SubFieldA'
            }
        },
        {
            Field: 'Field D',
            SubDocument:
            {
                FieldOfSubDocument:'SubFieldC'
            }
        }
    ]
}
```

Figure 15 Many-to-Many Document

The tables in Figure 16 are derived from the document collection of Figure 15. The table generated for the parent document is called DocumentCollection, named for the name of the document collection, and a separate table for the subdocument, named for the field that contained it, SubDocument. Once more, a new ID will be automatically generated for each table to serve as the primary key. The field Field was generated on the

DocumentCollection table to mirror the parent document. FieldOfSubDocument was a field on the subdocument, and is therefore generated on the SubDocument table.

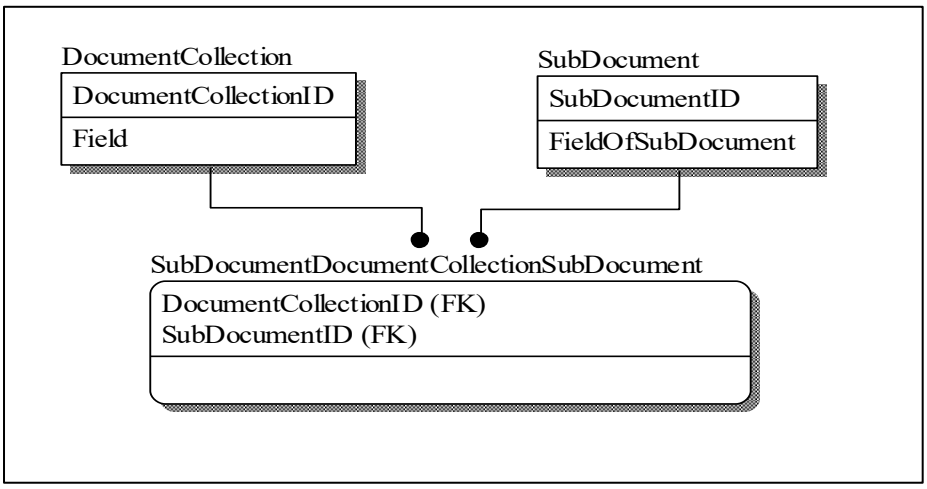

Figure 16 Many-to-Many Relationship Transformation

The primary keys of each table are copied to a new table, which will be the bridge table. The keys will provide a foreign key constraint to each of the tables, DocumentCollection and SubDocument, to form the many-to-many relationship. The two keys on the bridge table will become a composite primary key for the newly derived bridge table, meaning that the combination of the two must be unique.

In practice, bridge tables are usually named as a concatenation of the two tables it is bridging; the methodology follows that convention. The bridge table is named for the first table, followed by the second table. In Figure 16, the tables DocumentCollection and SubDocument are related through a bridge table, which is named DocumentCollectionSubDocument using this convention.

DocumentCollectionSubDocument contains a composite primary key made of the DocumentCollectionID and SubDocumentID foreign keys. When all three tables are joined together, the data of the document collection can be recreated.

3.4 Data Types

For anything within the document database to be transformed at the physical level, it must first be determined what exists within the collections of the document database. Through analysis of the data, it can be determined what data types exist in each field and if that field should be nullable within the derived relational physical schema.

With the discovered data types, if they are consistent, then an equivalent data type should be used within the generated schema. For instance, integer fields stored in a BIGINT column or a float field stored in a FLOAT column. An unknown data type is stored in a VARCHAR(MAX). Further details are found in Table 2.

Detected Data	SQL Data Type
String or XML	VARCHAR(MAX)
Floating-point	FLOAT(n)
Integer and Floating-point	FLOAT(n)
Date or Time	DATETIME
Bool or Bit	BIT
Unknown	VARCHAR(MAX)

Table 2 Data Type Conversion

3.5 Output

The output of the methodology will be the scripts for the creation of a database which would be able to receive the data from the document database. This database will have the subdocuments of the collection extracted out into additional tables to lessen the amount of null or duplicated data. With the use of normalization, the schema will provide a more efficient storage structure.

Chapter four describes how Variety, a schema analyzer for MongoDB, was utilized to determine the implicit schema of the document database, and the necessary modifications that were made to detect duplicate data. Chapter four will also describe how the

methodology was used for the transformation of a document database to a relational database. Chapter five presents the results of several tests for the conversion software. Chapter six provides conclusions and further ideas for future work.

Chapter 4

IMPLEMENTATION

The methodology described in chapter 3 generates a relational database schema from a document database. The schema generation is based upon the implicit schema of the document collection; however the methodology does not give a specific way by which the implicit schema should be derived. The information in the implicit schema consists of the arrangement of the fields and subdocuments, data types, and what fields contain duplicated values.

The implementation in this chapter offers a tool to solve the problem of retrieving the implicit schema. Variety is a schema generator for MongoDB written in Node [Dvorak16]. Variety takes in the name of the collection to be mapped, and then outputs a JSON object that describes the schema of the documents in that collection. Once the implicit schema is defined, the methodology described in chapter 3 generates a relational schema from a source document database.

Section 4.1 will cover how Variety works, and the structure that it creates. Section 4.2 introduces a solution to the one gap that Variety leaves in the metadata about the document database. Within section 4.3, the output of Variety is consumed by the system to generate the metadata to be used in table generation. Section 4.4 shows the final tweaks that need to be made before the system is ready to generate the schema. Sections

4.5 and 4.6 cover how the system generates scripts for table creation and foreign key relationships, respectively.

4.1 Variety

Variety is an open source software designed to determine the implicit schema of a document collection. Variety takes an input of a MongoDB connection, as well as some settings. Variety outputs a JSON object that describes the structure of the document collection.

Figure 17 contains a sample of the metadata that Variety outputs. The _id nested object provides a key which contains the object path of the field that it is describing in this object. The example features address.building and address.street. These examples indicate that from the root document, there is a subdocument called address. On that subdocument, address, are the fields: building and street.

The key is used as the identifier for the field being described by Variety. If Variety encounters an array datatype, it uses a placeholder string for the key of that field. The placeholder string acts as an anonymous object to differentiate between the field which contains an array and the unnamed document which exists within the array.

```
{
        "_id" : {
                "key" : "address.building"
        },
        "value" : {
                "types" : {
                        "String" : 25360
                }
        },
        "totalOccurrences" : 25360,
        "percentContaining" : 100
},
{
        "_id" : {
                "key" : "address.street"
        },
        "value" : {
                "types" : {
                        "String" : 25360
                }
        },
        "totalOccurrences" : 25360,
        "percentContaining" : 100
},
```


Figure 17 Variety Output Sample

Next, the value property of the output sample describes the data types that exist in the field being described. In the sample, both address.building and address.street contain 25360 instances of data of the type String. The properties in the types object describe the data types. Variety relies on the JavaScript language function typeof to determine the data type. Table 3 lists the data types in JavaScript and the return value from typeof function [Mozilla17B]. Therefore, anything in the result column is a valid property name on the types object.

Type	Result
Undefined	"undefined"
Null	"object"
Boolean	"boolean"
Number	"number"
String	"string"
Symbol (new in ECMAScript 2015)	"symbol"
Function object (implements [[Call]] in ECMA-262 terms)	"function"

Table 3 JavaScript Data Types

Variety derives this information by extracting all of the data out of the targeted MongoDB collection, and then running a reduce on it. Reduce is a method in JavaScript that will take a collection of data and perform an action on every item to generate one single return object that in some way represents the entire collection [Mozilla17A].

The reduce iterates over each document in the collection, processing it and updating the metadata that will become the output information. During each iteration, Variety determines the data type of the field in the particular document and either adds it to the metadata or increments the counter. When the reduce completes, the output is a fully-

formed set of metadata. The metadata describes the structure of the document collection. The metadata includes object graphs, nullability, and datatypes.

The methodology requires an input with data about the document structure, data types, nullability, and where duplication exists. The output from Variety can be interpreted to obtain all of these required inputs, except for if duplication exists. Metadata about where duplication exists is important because it will be used by the methodology to determine if a many-to-one or many-to-many relationship exists.

Variety is an open source software under an MIT license, which allows for distribution and modification of the source code [OpenSource.org17]. With some minor modifications to Variety, the required information about duplicated subdocuments can be combined with Variety's already useful output.

4.2 Duplication Detection

During the reduction of the collection, the system tracks all previous values for each field. Upon the first duplicate being encountered, the field is flagged as having duplicates and the cache of previous values is dumped. The dump is to attempt to preserve memory as it is possible for the data being processed to be large.

Once the field is flagged as having duplicates, it is no longer checked. The code for tracking duplicate values is depicted in Figure 18. If no previous duplication has been

found, the current value being processed is checked against the array of previous values. If the value is a duplicate, then the field is marked as having duplicates in the duplicationTracking array. If the value is not a duplicate, the value for the field is inserted into the values array.

```
if(duplicationTracking[key] === false){
    if(values[key].values.indexOf(value.toString()) >= 0){
        duplicationTracking[key] = true;
        values[key].values = [];
    }else{
        values[key].values.push(value.toString())
    }
}
```

Figure 18 Tracking of Duplicates

Later, when the metadata is being used to generate Variety's output, the duplicationExists property is added to the output. The duplicationExists property exists for each field in the Variety metadata, allowing the system to determine if the document as a whole is duplicated in the collection. If all fields on a document has the duplicationExists property set to true, then the system will treat that document as being able to have duplicates. The duplication represented impacts the kind of relationship chosen by the methodology. The addition of the duplicationExists property will alter the output of Variety from the structure in Figure 17 to that of Figure 19. Every field object has the duplicationExists property after the modifications.

```
{
        "_id" : {
                "key" : "address.building"
        },
        "value" : {
                "types" : {
                        "String" : 25360
                }
        },
        "totalOccurrences" : 25360,
        "percentContaining" : 100,
        "duplicationExists" : true
},
{
        "_id" : {
                "key" : "address.street"
        },
        "value" : {
                "types" : {
                        "String" : 25360
                }
        },
        "totalOccurrences" : 25360,
        "percentContaining" : 100,
        "duplicationExists" : true

},
```


Figure 19 Variety Output with Duplication Detection

In cases where a document is in a many-to-one or many-to-many relationship, duplicationExists will be true on all fields. When the document is in a one-to-one or one-to-many relationship, all fields will have duplicationExists set to false.

4.3 New Development

The system begins by executing the modified version of Variety and receiving the JSON structure in Figure 19. Upon return of the completion of Variety's run, Figure 20 depicts the code that is executed. This consists of four methods which are executed in sequence

to generate the schema: processField, refineRelationships, generateTableCreationSql, and generateRelationshipSql. At the end of processing, the createScript and alterScript are concatenated, and sent to the connected database.

```
function postPromise(){
    var schemaAnalysis = JSON.parse(global.varietyResult);
    schemaAnalysis.forEach(processField);
    refineRelationships();
    var createScript = generateTableCreationSql();
    var alterScript = generateRelationshipSql();
}
```

Figure 20 Implemented Methodology

Section 4.4 will cover what happens during the processField method. The refineRelationships method which determines the cardinality of the relationships will be covered in section 4.5. Section 4.6 will cover the generateTableCreationSql method. Section 4.7 will explain the generateRelationshipSql method.

4.4 Relationship Structure Generation

The processField method is responsible for most of the heavy lifting of the system. The method will execute on each field that Variety provides metadata for, and generate new metadata that describes the tables and relationships.

The core of the method processField is embodied by the pseudocode in Figure 21. Due to the flat structure that Variety outputs, the system relies upon the _id.key of each field.

The identifier in the _id.key field contains the full path to the field, so all but the last item in the path are documents, and therefore will be processed into tables. If it already exists as a table, it is not changed.

```
Traverse varietyResult
        Split _id.key on .
        Check that each part but the last already exists in table
Array
                If not exist, create
                Create Relationship
        Determine Data Type
        Check Duplications
        Add field to table
```

Figure 21 processField Function Overview

There is one exception to the handling of the _id.key: the array placeholder. If the system encounters the array placeholder which Variety outputs in place of an array, the system moves on to the following name and notates that it is an XToMany relationship. The selection of XToMany coincides with the methodology, the scenarios where the subdocument exists in an array, leading to either a one-to-many or many-to-many relationship.

If the array placeholder was not encountered in the object path immediately before the table being processed, the table is noted as an XToOne relationship. If that array is not present, it is impossible for there to be multiple subdocuments for that document. This relationship type is not noted directly on the data object that tracks the tables and the columns that the fields will become, rather relationships are stored separately. Figure 22

shows the object that represents a relationship. Each relationship is defined by the two tables it relates, and the type of relationship. The table names are stored in the order that they appear in the key of the Variety metadata. The order of the table names will always have the parent document's name be in the firstTable field, while the secondTable will always be the subdocument.

```
{
    firstTable: firstTableName,
    secondTable: secondTableName,
    relationshipType = RelationshipType.XToOne
}
```

Figure 22 Relationship Object

At the time the relationship object is created, at the same time as the table, it is unknown if duplications of the subdocument being represented can exist. At the time that the relationship object is generated, it is only known whether the subdocument is inside an array or not. The existence of the subdocument as a field or item in an array is captured by the XToMany and XToOne relationship types, shown in Figure 23. If the subdocument is contained within an array, it is stored as an XToMany relationship. The XToMany relationships will be derived into either a one-to-many or many-to-many relationship based upon duplication. If the subdocument is not contained within an array, then it is stored as an XToOne relationship. The XToOne relationship is an intermediary step towards either a one-to-one or many-to-one relationship. The relationship refining step, which will be discussed in section 4.5, will transform these intermediary relationships to their final types.

```
var RelationshipType = {
    OneToOne: 1,
    OneToMany: 2,
    ManyToOne: 3,
    ManyToMany: 4,
    XToMany: 5,
    XToOne: 6
}
```

Figure 23 Relationship Types

Following the creation of any necessary tables, the field is analyzed to determine its data type. The system does a very trivial data type conversion. As previously covered, the Variety output contains an object with all possible data types. The premise that the system follows is to select the most general data type. Therefore, if any data types are of type string, a string is used. JavaScript has no innate way to detect a float versus an integer, so a float will always be used. If dissimilar data types are used, the system will default to using a string.

The selected data type is then put in an object which contains information about the field, referred to as the fieldInformation. The fieldInformation object will also track the nullability of the field, also derived from the Variety metadata, and whether duplications exist in that field. The full structure is outlined in Figure 24. The fieldInformation object is associated with the table object for easier processing later.

```
var fieldInformation = {
    name: fieldName,
    dataType: dataType, // dataType to be used in the SQL
    isNullable: field.percentContaining !== 100,
    hasDuplications: field.duplicationExists
};
```

Figure 24 fieldInformation Object

When the metadata for the table is first created, it is assumed that the table can have duplications. As each field is processed, this assumption is checked against the modified Variety output. Once a field is found that there does not exist a duplicate for, the metadata for the entire table is corrected as not having any duplicates.

4.5 Refine Relationships

Once the fields have been processed, the system must refine the relationships that were previously only partially derived. As covered in section 4.3, at the time of field processing, the relationship could only be partially determined. By this time, all duplication has been processed, and the final relationship types can be determined. Following that, each table will then have its script generated.

Refining the relationship becomes an interesting problem. Duplication is assumed to occur on each table, until there is a single field found which does not have duplication. It is possible in a scenario of deeply nested subdocuments, that the deepest level is the only field that is not duplicated. Therefore, this would need to be propagated up all

relationships to ensure that none of the parent documents are marked as having duplicates.

The recursive method propogateDuplication, shown in Figure 25, marks all ancestors of a document as not having duplicates when a subdocument is determined to not have duplications. A table name is passed into the method, and is used to find the table in the tableArray. Once the table has been found, it is corrected to be marked as having no duplicates. Then, the parent table is located, and is passed into the propogateDuplication function again. Every table that does not have duplicates has propogateDuplication called on it.

```
function propagateDuplication(tableName){
    var firstTable = findTable(tableName);
    firstTable.hasDuplications = false;
    var relationshipsToModify = relationships.every(
        function(rel){ return rel.secondTable === tableName });
    for(var y = 0; y < relationshipsToModify.length; y++){
        var relationship = relationshipsToModify[y];
        propagateDuplication(relationship.firstTable);
    }
}
```

Figure 25 propagateDuplication Function

Once the duplication flag has been propagated properly, the true relationship types, such as one-to-one or many-to-one, can be determined. Based on the previously partially set relationship type flag of either XToOne or XToMany and the newly set duplication flag, the system determines the true relationship between the two tables. Table 4 shows how the intermediary relationship type is transformed into the true relationship type. If it was

previously determined that the intermediary relationship type was XToOne and duplication is present, then the relationship is a many-to-one relationship. If the intermediary relationship type is XToOne and duplication is not present, then the relationship must be a one-to-one relationship. If the intermediary relationship type was XToMany, and duplication is present, the relationship is determined to be many-to-many. The final scenario is that of an XToMany intermediary relationship type with no duplication, which is determined to be a one-to-many relationship. The relationship type will be used in section 4.6 and section 4.7 to determine how the foreign key relationships are generated.

	XToOne	**XToMany**
No Duplication	One-to-One	One-to-Many
Has Duplication	Many-to-One	Many-to-Many

Table 4 Relationship Type Derivation

4.6 Table Script Generation

After all of the metadata about the tables has been generated, the system proceeds into the script generation phase. The generation of the scripts for table creation and key creation has been broken into separate steps. This was done to avoid having to determine the correct order to generate the tables so that the foreign keys would always have tables to reference upon creation. By creating the tables and then the key relationships, order of creation no longer matters.

Figure 26 describes how the scripts to create the tables are generated. The table is constructed of a few parts which are fitted together. First the actual create table statement, and the newly generated primary key column. Second, all the fields that were associated with the table are added to the script. At this point, the system also utilizes the previously determined data type and nullability. Finally, the foreign keys are appended, and the statement terminated.

```
Traverse tables Array
    Create name + "Id IDENTITY(1,1) PRIMARY KEY"
    Iterate fields
        Name + " " + dataType
    Find all relationships with this table in first as One-to-Many OR
second in One-to-One or Many-to-One
        Add FK for each
```

Figure 26 Table Script Generation Overview

To determine which foreign keys are necessary, the system searches the list of relationships. In cases where the table is the first table of a one-to-many relationship, or the second table in a one-to-one or a many-to-one relationship, the foreign key is generated. This will facilitate the relationship present in the document database, as discussed in Chapter 3.

4.7 Relationship Script Generation

After the tables have been generated, the relationships can be generated based on the relationship metadata that was created. In this case, the relationship list is traversed.

Each relationship contains the firstTable and secondTable, as well as the relationshipType. The fields are used by the code in Figure 27 to generate the foreign key relationships between the tables.

```
function generateRelationshipScript(relationship){
    var starter = alterRelationshipsSql === "" ? "\n\n" :
alterRelationshipsSql + "\n\n";

    if(relationship.relationshipType === RelationshipType.OneToOne){
        alterRelationshipsSql = starter + "-- One-to-One\n" +
createForeignKey(relationship.firstTable, relationship.secondTable);
    }

    else if(|| relationship.relationshipType ===
RelationshipType.ManyToOne){
        alterRelationshipsSql = starter + "-- Many-to-One\n" +
createForeignKey(relationship.firstTable, relationship.secondTable);

    else if(relationship.relationshipType ===
RelationshipType.OneToMany){
        alterRelationshipsSql = starter + "\n-- One-to-Many \n" +
createForeignKey(relationship.secondTable, relationship.firstTable);
    }

    else if(relationship.relationshipType ===
RelationshipType.ManyToMany){
        alterRelationshipsSql = starter + "\n-- Many-to-Many\n" +
generateManyToManyConstraint(relationship);
    }
}
```

Figure 27 generateRelationshipScript Function

In a one-to-one or many-to-one relationship, the foreign key will be added to the firstTable. In a one-to-many relationship, the foreign key will be added to the secondTable. The createForeignKey function generates the key, specifically from the table name in the first parameter referring to the table name in the second parameter.

The variant from this list of alter scripts is when a many-to-many relationship is encountered. For the many-to-many relationships, a bridge table must still be generated. The bridge table generation is handled in the function, generateManyToManyConstraint, depicted in Figure 28. The bridge table is named as a concatenation of the names of the two tables it joins, and it contains the primary key of each of those tables. The foreign key relationships are created at the same time as the table because the script for all non-bridge tables has already been created. Finally, a composite primary key is created for the bridge table.

```
function generateManyToManyConstraint(relationship){
    // Create Bridge Table
    var starter = "\n\n"

    var createTable = "CREATE TABLE " + relationship.firstTable +
relationship.secondTable;
    var firstColumn = relationship.firstTable + "Id INT REFERENCES " +
relationship.firstTable + "(" + relationship.firstTable + "Id)";
    var secondColumn = relationship.secondTable + "Id INT REFERENCES "
+ relationship.secondTable + "(" + relationship.secondTable + "Id)";
    var compositeKey = "CONSTRAINT PK_" + relationship.firstTable +
relationship.secondTable + " PRIMARY KEY (" + relationship.firstTable
+ "Id," + relationship.secondTable + "Id)";

    var createBridgeTableSql = starter + createTable + "(\n" +
firstColumn + ",\n" + secondColumn + ",\n" + compositeKey + "\n)";
    return createBridgeTableSql;
}
```

Figure 28 generateManyToManyConstraint Function

4.8 Output

Once script generation has completed, the transformation of the document database has completed. The scripts for table generation and adding the key relationships is finished. The system can either output the scripts or send them directly to a relational database. The system implemented currently does a combination of outputting the scripts to command line and runs against a SQL Server instance, though Node supports a wide variety of relational databases. Figure 29 is a sample of the script output of the system, which would be run against the target database system.

```
CREATE TABLE OneToOne(
OneToOneId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
uniqueField FLOAT(12) NOT NULL,
uniqueChildId INT
)

CREATE TABLE uniqueChild(
uniqueChildId INT IDENTITY(1,1) PRIMARY KEY,
uniqueNumber FLOAT(12) NOT NULL
)

-- One-to-One Relationship
ALTER TABLE OneToOne
ADD CONSTRAINT FK_OneToOne_uniqueChild
FOREIGN KEY (uniqueChildId)
REFERENCES uniqueChild(uniqueChildId)
```

Figure 29 Sample Script Output

The sample contains a one-to-one relationship between a parent document and a subdocument contained by a field named uniqueChild. The parent document contained a field, uniqueField, which only contained numeric values. The subdocument,

uniqueChild, contains a field, uniqueNumber, which only contained numeric values. Both, uniqueField and uniqueNumber, were generated as FLOATs. The foreign key relationship is established by the uniqueChildId being on the OneToOne table. The alter statement adds the foreign key constraint to relate the two tables using the uniqueChildId field on both tables.

Chapter 5

TRANSFORMATION COMPARISONS

Technology can often be divided into the safe technologies, and those on the bleeding edge. When a new technology fails to work as expected or is outgrown, ideally there is a route to get to a safer technology. Prior to the described methodology, the route from a document database to a relational database was a difficult endeavor.

The methodology seeks to ease the transition from a document database to relational database system. The methodology automatically generates a relational database schema that can be used to store the data from the document database in a more normalized structure. With the relational schema generated based upon the structure of the data in the document database, the transition process from document database to relational database is simplified.

Section 5.1 will describe how the system performs with one-to-one relationships. Sections 5.2, 5.3, and 5.4 will continue with one-to-many, many-to-one, many-to-many relationships, respectively. Section 5.5 will bring these all together to explain how the system handles multi-layer documents.

Each section will also provide evidence of the accuracy of the conversion through analysis of the original document compared to a JSON-serialized DTO created from the

newly generated database tables. As discussed in chapter 1, document databases have an implicit schema, the structure of the database is determined by the application that uses it, rather than by the database itself. Applications use plain objects referred to as DTOs to stage data and save it to the database. One of the tools that can be used to create DTOs is Microsoft's Entity Framework. Through use of Microsoft Entity Framework, DTOs can be generated from a relational schema.

To evaluate the methodology, a relational database is generated by the implementation discussed in chapter 4. From the generated relational database, Entity Framework is used to generate DTOs representing the database tables. These DTOs are then populated with data and serialized into JSON. The final serialized content of the DTOs is compared to the original data stored in the document that is being transformed.

5.1 One-to-One Relationships

A one-to-one relationship is detected by the system when a document contains a field that only holds one subdocument, and that document is not duplicated anywhere else in the collection. As demonstrated in chapter 2, one-to-one relationship can be facilitated through either table having the key of the other table, or the two tables sharing the key. In the methodology, the key of the subdocument is placed on the parent document.

Figure 30 contains a document that was supplied to the system to generate a one-to-one relationship. The document contains a field, uniqueField, which is unique across all

documents in the collection. uniqueChild is a field which contains a unique subdocument. That subdocument contains a field called uniqueNumber, which contains a unique numeric value.

```
{
    "_id" : ObjectId("58379145df093fc9ca9cf94b"),
    "uniqueField" : 12,
    "uniqueChild" :
    {
        "uniqueNumber" : 72
    }
}
```

Figure 30 One-to-One Sample Document

The system utilizes the methodology to generate the script in Figure 31 for a one-to-one relationship. The script creates a table corresponding to each document, with the same fields. The OneToOne table has the uniqueField derived from the parent document. The uniqueChild table has the uniqueNumber field from the uniqueChild subdocument. The uniqueNumber field is of float type to capture the numeric value. The uniqueChildId is added to the OneToOne table, and the alter script uses it to form the foreign key relationship, as described in the methodology.

```
CREATE TABLE OneToOne(
OneToOneId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
uniqueField FLOAT(12) NOT NULL,
uniqueChildId INT
)

CREATE TABLE uniqueChild(
uniqueChildId INT IDENTITY(1,1) PRIMARY KEY,
uniqueNumber FLOAT(12) NOT NULL
)

-- One-to-One Relationship
ALTER TABLE OneToOne
ADD CONSTRAINT FK_OneToOne_uniqueChild
FOREIGN KEY (uniqueChildId)
REFERENCES uniqueChild(uniqueChildId)
```

Figure 31 One-to-One Generated Script

When the generated tables are input into Entity Framework, the resulting classes are depicted in Figure 32. The diagram separates the properties of the classes into two categories: properties and navigation properties. Properties are the standard fields which were imported from the relational database. Navigation properties are special fields that Entity Framework automatically generates to facilitate relationships. The navigation property usually takes the form of a single, or an array, of the related object. Whether the property contains a single object or an array of objects facilitates the relationship of the generated class to one or many of the other class.

The classes generated in Figure 32 support a one-to-many relationship, as evidenced by the OneToOnes navigation property on the uniqueChild class. A one-to-many relationship is less strict than the one-to-one that was originally represented in the MongoDB, but can still be used to form the same logical relationship.

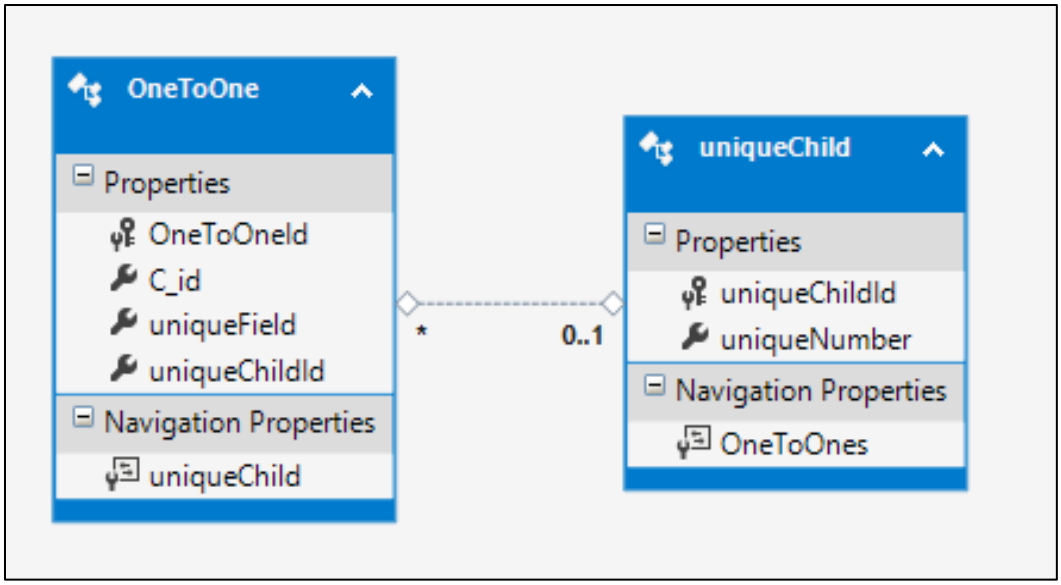

Figure 32 One-to-One DTOs

The true test is if the DTOs serialize into a structure that looks similar to that of the data stored in MongoDB. Table 5 demonstrates the JSON of the original MongoDB Document and the JSON of a serialized DTO that had been generated by Entity Framework. The implicit schema of the MongoDB is driven by the application's serialized DTOs, such as those generated by Entity Framework.

MongoDB Document	Serialized DTO

<pre>{
    "_id" :
ObjectId("58379145df093fc9ca9cf94b"),
    "uniqueField" : 12,
    "uniqueChild" :
    {
        "uniqueNumber" : 72
    }
}</pre>
	<pre>{
	    "OneToOneId": 0,
	    "C_id":
	"58379145df093fc9ca9cf94b",
	    "uniqueField": 12.0,
	    "uniqueChildId": 0,
	    "uniqueChild": {
	        "uniqueChildId": 0,
	        "uniqueNumber": 72.0,
	        "OneToOnes": []
	    }
	}</pre>

Table 5 Serialized One-to-One DTO Comparison

In the serialized DTO, the structure is the same as the original document. The uniqueField is on the parent document and the parent object. There is a subdocument contained in the uniqueChild field which contains the uniqueNumber field. In the serialized DTO, these fields appear in the same places.

The most notable difference between the original document and generated JSON is the presence of a OneToOnes array on the generated JSON. The OneToOnes array would be expected to a singular OneToOne object. The appearance of an array is due to the methodology's choice of placing the uniqueChild table's id on the OneToOne table as a foreign key, as opposed to having the two tables share a single key. Also, it is noticeable that the integer fields are now represented by floats due to the simple generalization of data types that the implementation utilizes. The OneToOneId and uniqueChildId were added during the methodology, and are therefore absent from the source document, but are present in the serialized DTO.

5.2 One-to-Many Relationships

A one-to-many relationship is discovered by the array that stores unique subdocuments. The methodology facilitates the one-to-many relationship by generating tables for both documents. The table representing the subdocument then has the id of the document appended to it. The shared key from the document facilitates the foreign key relationship.

Figure 33 presents a document from the source MongoDB which should generate a one-to-many relationship. The document has a uniqueField and uniqueChildren fields. The uniqueChildren field contains an array of subdocuments. The array of subdocuments is the cause of the one-to-many relationship between the parent document and the uniqueChildren subdocuments. Each subdocument has the uniqueNumber field, which is a number that is not repeated within the collection of documents.

```
{
    "_id" : ObjectId("5837913bdf093fc9ca9cf949"),
    "uniqueField" : 12,
    "uniqueChildren" :
    [
        { "uniqueNumber" : 72 },
        { "uniqueNumber" : 43 }
    ]
}
```

Figure 33 One-to-Many Sample Document

The system takes as input a document collection, which contains the document from Figure 33, and generates the SQL script depicted in Figure 34. The script creates a pair of tables, OneToMany and uniqueChildren. The OneToMany table contains a uniqueField column, just as can be seen in the parent document of the table. The uniqueChildren table contains a uniqueNumber numeric field, and a OneToManyId. The OneToManyId is then linked back to the OneToMany table in a foreign key constraint by the alter table statement.

```
CREATE TABLE OneToMany(
OneToManyId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
uniqueField FLOAT(12) NOT NULL
)

CREATE TABLE uniqueChildren(
uniqueChildrenId INT IDENTITY(1,1) PRIMARY KEY,
uniqueNumber FLOAT(12) NOT NULL,
OneToManyId INT
)

-- One-to-Many
ALTER TABLE uniqueChildren
ADD CONSTRAINT FK_uniqueChildren_OneToMany
FOREIGN KEY (OneToManyId)
REFERENCES OneToMany(OneToManyId)
```

Figure 34 One-to-Many Generated Script

Comparing the database schema to the sample data that generated it, they are expectedly similar. The foreign key constraint allows for there to be multiple uniqueChildren records per OneToMany record, which follows the data having an embedded array. This becomes more apparent with further analysis of the generated table.

The DTOs generated by Entity Framework can be seen in Figure 35. Entity Framework notes that there exists a one-to-many relationship between the OneToMany and uniqueChildren objects, as evidenced by the "0..1" and asterisk on the line between the object diagrams. One interesting consequence of the Entity Framework generation is the uniqueChildrens array, with the incorrect "s" on the end. The additional "s" is due to a convention in Entity Framework that automatically appends an "s" to properties which are arrays. The extra pluralization can be fixed by editing the DTO to match the original MongoDB structure, or through changing the default settings of Entity Framework. For the purpose of analyzing the methodology, the Entity Framework settings were intentionally left unaltered.

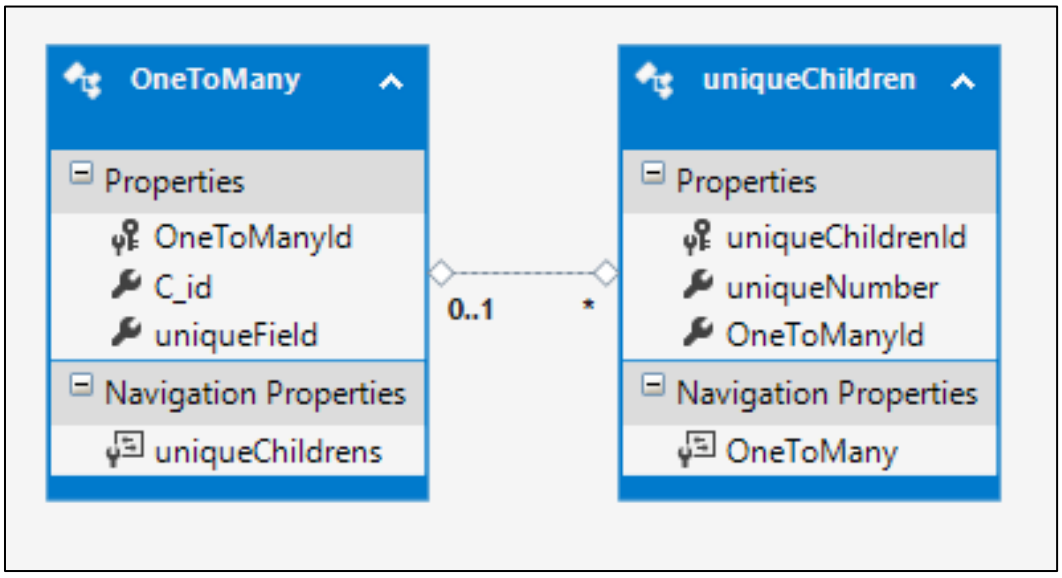

Figure 35 One-to-Many DTOs

Table 6 shows a comparison of the source document and the final serialized DTO. When comparing to the original structure, it appears the same. The OneToMany object has a

navigation property of uniqueChildrens, which is an array of uniqueChildren. Conversely, uniqueChildren has a navigation property of OneToMany, which is a single instance of the OneToMany object. The properties being generated in this way shows the one-to-many relationship more clearly.

MongoDB Document	**Serialized DTO**

<pre>{
    "_id" :
ObjectId("5837913bdf093fc9ca9cf949"),
    "uniqueField" : 12,
    "uniqueChildren" :
    [
        { "uniqueNumber" : 72 },
        { "uniqueNumber" : 43 }
    ]
}</pre>
	<pre>{
	    "OneToManyId": 0,
	    "C_id":
	"5837913bdf093fc9ca9cf949",
	    "uniqueField": 12.0,
	    "uniqueChildrens":
	    [
	        {
	            "uniqueChildrenId":
	0,
	            "uniqueNumber":
	72.0,
	            "OneToManyId": 0,
	            "OneToMany": null
	        },
	        {
	            "uniqueChildrenId":
	1,
	            "uniqueNumber":
	43.0,
	            "OneToManyId": 0,
	            "OneToMany": null
	        }
	    ]
	}</pre>

Table 6 Serialized One-to-Many DTO Comparison

5.3 Many-to-One Relationships

A many-to-one relationship is the first relationship in which duplication plays a role. The many-to-one relationship is discovered by the methodology when a document has a field containing a subdocument which is duplicated across other documents. Figure 36 contains two documents from a collection of documents that has a many-to-one relationship. Each document contains a unique fieldA and oneDuplicated field.

```
{
    "_id" : ObjectId("58377ecedf093fc9ca9cf943"),
    "fieldA" : "stuff",
    "oneDuplicated" :
    {
        "sub" : "dupe"
    }
},
{
    "_id" : ObjectId("58377ecedf093fc9ca9cf944"),
    "fieldA" : "others",
    "oneDuplicated" :
    {
        "sub" : "dupe"
    }
}
```

Figure 36 Many-to-One Sample Document

The oneDuplicated field contains a single subdocument. The subdocument contains a sub field, sub, which is duplicated between two documents in the collection. Because the sub field of the oneDuplicated subdocument is duplicated, the methodology considers the entire oneDuplicated object to be duplicated. The duplication of the subdocument indicates a many-to-one relationship, as described in the methodology.

The system generates a script, as depicted in Figure 37, to create the tables. There is a ManyToOne table and a oneDuplicated table. ManyToOne has an _id and fieldA columns. The oneDuplicated table has a field called sub. The oneDuplicatedId is copied to the parent document, and is referenced in the foreign key constraint between the ManyToOne and oneDuplicated tables, which is created by the alter table statement.

```
CREATE TABLE ManyToOne(
ManyToOneId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
fieldA VARCHAR(MAX) NOT NULL,
oneDuplicatedId INT
)

CREATE TABLE oneDuplicated(
oneDuplicatedId INT IDENTITY(1,1) PRIMARY KEY,
sub VARCHAR(MAX) NOT NULL
)

--  Many-to-One Relationship
ALTER TABLE ManyToOne
ADD CONSTRAINT FK_ManyToOne_oneDuplicated
FOREIGN KEY (oneDuplicatedId)
REFERENCES oneDuplicated(oneDuplicatedId)
```

Figure 37 Many-to-One Script

Once the tables are generated into C# classes by Entity Framework, they produce the classes in Figure 38. Entity Framework detected the relationship between the two objects, generating the navigation properties and the line between them. As in the case of the one-to-many in Figure 35, the relationship is annotated by an asterisk and a “0..1”, except in opposite sides than the previous example. The relationship represents a many-to-one relationship, whereas the previous was a one-to-many relationship.

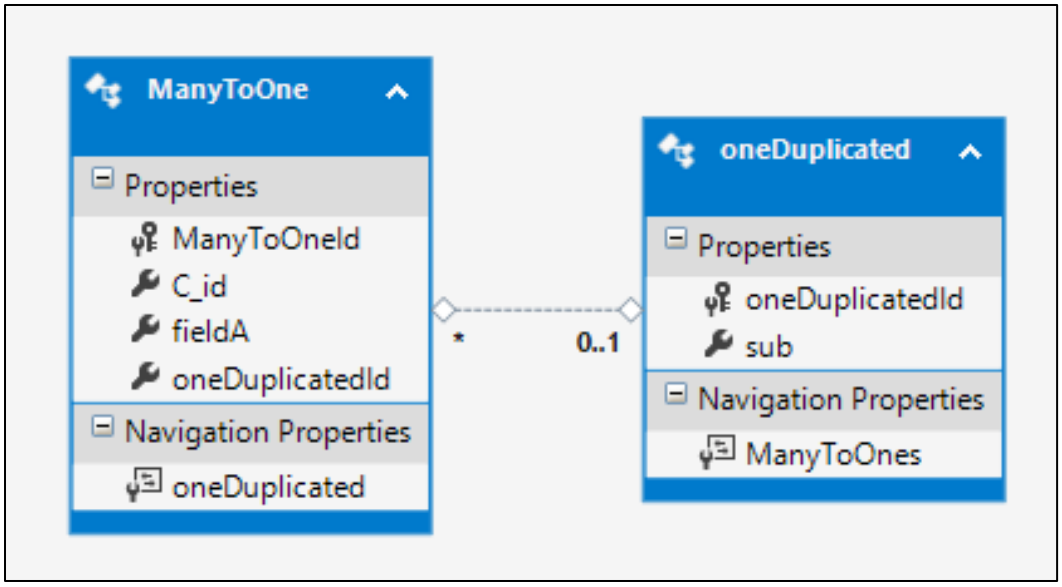

Figure 38 Many-to-One DTOs

Once the DTO is serialized with data, the result is depicted in Table 7. The parent document has fieldA, the parent object has fieldA. The original data from MongoDB featured the oneDuplicated field as a subdocument of the document. The field oneDuplicated appears as a single subobject of the parent object. The field sub is on the subdocument, oneDuplicated, in the original document. The generated object has the field sub on the onDuplicated subobject.

MongoDB Document	Serialized DTO

<pre>{
    "_id" :
ObjectId("58377ecedf093fc9ca9cf943"),
    "fieldA" : "stuff",
    "oneDuplicated" :
    {
        "sub" : "dupe"
    }
}</pre>
	<pre>{
	    "ManyToOneId": 0,
	    "C_id":
	"58377ecedf093fc9ca9cf943",
	    "fieldA": "stuff",
	    "oneDuplicatedId": 0,
	    "oneDuplicated":
	    {
	        "oneDuplicatedId": 0,
	        "sub": "dupe",
	        "ManyToOnes": []
	    }
	}</pre>

Table 7 Serialized Many-to-One DTO Comparison

The source and target have the same structure of fields as previous, but the serialized DTO shares an additional piece of information about the structure. In the original document, duplication can only be determined by scanning the entire document collection. In the serialized DTO, the duplication emerges as the ManyToOnes field, which contains an array. The extra field is a side-effect of the navigation properties that Entity Framework automatically generates. The ManyToOnes field also provides insight into how the subobject can relate to many of the parent, thus confirming the many-to-one relationship.

5.4 Many-to-Many Relationships

A many-to-many relationship occurs when a document has an array of subdocuments, which are duplicated between documents. The many-to-many relationship is unique in

that it requires a bridge table to be generated for the relationship to be modelled in a relational database.

Figure 39 shows an example of a document that contains a many-to-many relationship in MongoDB. The manyDuplicated field holds an array of subdocuments. Within the array of subdocuments, at least one of the subdocuments is a duplicate of one in another document. In the scenario presented by the example, the document collection would contain a second document which in its manyDuplicated field has a subdocument that also has a field sub with the value “dupe”. The subdocument must be an exact match for it to be considered a duplicate.

```
{
    "_id" : ObjectId("58377e48df093fc9ca9cf941"),
    "fieldA" : "value",
    "manyDuplicated" :
    [
        {
            "sub" : "dupe"
        },
        {
            "sub" : "not"
        }
    ]
}
```

Figure 39 Many-to-Many Sample Document

The schema generated by the system, based upon the methodology is displayed in Figure 40. In the previously analyzed relationships, two tables and a constraint are generated. In this example, three tables are generated. The ManyToMany table represents the parent document, with fieldA taken from that document. ManyToManyId is the automatically

generated unique identifier. The second table, manyDuplicated, represents the subdocument that was contained within the field called manyDuplicated. manyDuplicatedId is the automatically generated unique identifier, and sub is the field from the subdocument.

```
CREATE TABLE ManyToMany(
ManyToManyId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
fieldA VARCHAR(MAX) NOT NULL
)

CREATE TABLE manyDuplicated(
manyDuplicatedId INT IDENTITY(1,1) PRIMARY KEY,
sub VARCHAR(MAX) NOT NULL
)

-- Many-to-Many
CREATE TABLE ManyToManymanyDuplicated(
ManyToManyId INT REFERENCES ManyToMany(ManyToManyId),
manyDuplicatedId INT REFERENCES manyDuplicated(manyDuplicatedId),
CONSTRAINT PK_ManyToManymanyDuplicated PRIMARY KEY
(ManyToManyId,manyDuplicatedId)
)
```

Figure 40 Many-to-Many Generated Script

Where the sample in Figure 40 departs from the previous relationships is the third table. The ManyToManymanyDuplicated table is the bridge table that joins the ManyToMany and manyDuplicated tables. Section 2.1.3 focused on how a bridge table is required to facilitate a many-to-many relationship in a relational database. The name of the table is generated by concatenating the names of the two tables that are participating in the many-to-many relationships. The two fields on the ManyToManymanyDuplicated table are the ManyToManyId and manyDuplicatedId. Each of the two fields has a foreign key relationship to its source table: ManyToManyId to ManyToMany and manyDuplicatedId

to manyDuplicated. The primary key of the ManyToManymanyDuplicated is a composite key of the two fields, ManyToManyId and manyDuplicatedId. Setting up the primary key as a composite key guarantees uniqueness in the relationships between the tables.

This generated schema used as an input for Entity Framework generates the DTOs in Figure 41. Rather than generating three classes, to represent each table, Entity Framework generated two tables, and then abstracts the bridge table into a pair of lists in the navigation properties. The ManyToMany object has a manyDuplicateds array navigation property. The manyDuplicated object has a ManyToManies array navigation property. The manyDuplicateds and ManyToManies properties represent the two sides of the many-to-many relationship, so each object can relate to many of the other.

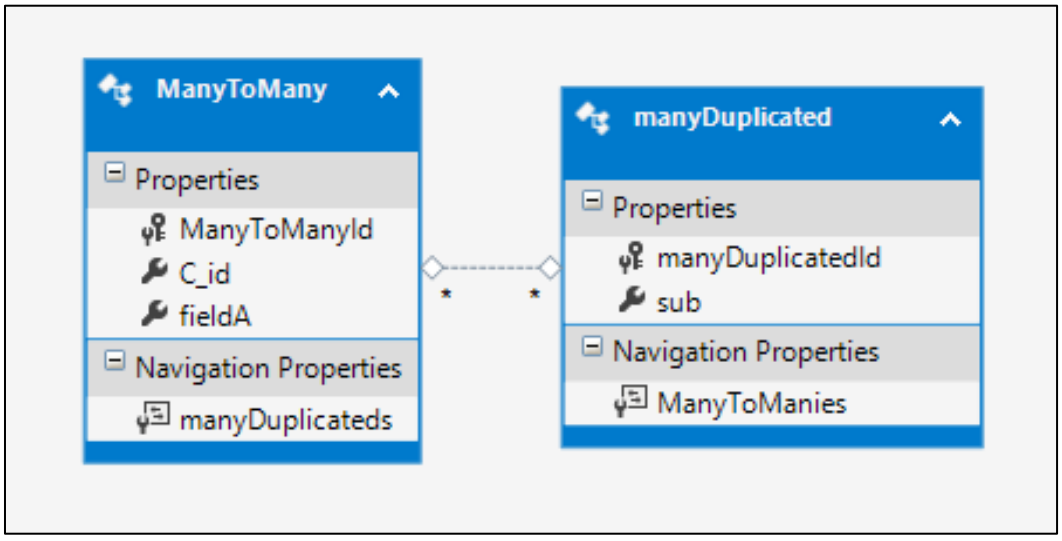

Figure 41 Many-to-Many DTOs

When the DTOs generated with Entity Framework are serialized, it generates the JSON object seen in Table 8. The parent document has a field A, just as the parent object in the serialized DTO. The parent document has a manyDuplicated field which contains an array of subdocuments. The parent object of the DTO contains a manyDuplicateds field which has an array of subobjects. In the MongoDB source, the subdocuments of the array had a single field, sub. The resulting DTO subobjects have the sub field, as well. The DTO subobjects also have the automatically generated manyDuplicatedId and the ManyToManies navigation property.

MongoDB Document	**Serialized DTO**

<pre>{
    "_id" :
ObjectId("58377e48df093fc9ca9cf941"),
    "fieldA" : "value",
    "manyDuplicated" :
    [
        {
            "sub" : "dupe"
        },
        {
            "sub" : "not"
        }
    ]
}</pre>
	<pre>{
	    "ManyToManyId": 0,
	    "C_id":
	"58377e48df093fc9ca9cf941",
	    "fieldA": "value",
	    "manyDuplicateds":
	    [
	        {
	            "manyDuplicatedId":
	0,
	            "sub": "dupe",
	            "ManyToManies": []
	        },
	        {
	            "manyDuplicatedId":
	0,
	            "sub": "not",
	            "ManyToManies": []
	        }
	    ]
	}</pre>

Table 8 Serialized Many-to-Many DTO Comparison

The ManyToManies property in the serialized DTO reinforces the idea of the many-to-many relationship. The ManyToManies property is an array of the parent object. Therefore, the parent object can contain many of the child with the manyDuplicateds property, while the child is able to contain many of its parents in the ManyToManies field. The many-to-many relationship is depicted by the manyDuplicateds and ManyToManies properties. The final serialized DTO reflects the structure of the data in the source MongoDB.

5.5 Duplication in Depth

Sections 5.1 through 5.4 explored the conversion of each relationship type from MongoDB into relational schema in SQL Server, followed by serialization of the DTOs as JSON objects. This section will be demonstrate how the system works with layers of duplication.

The collection DeepDuplication, depicted in Figure 42, has multiple layers of duplication. The collection consists of three tiers, DeepDuplication, sub, and alsoDuplicated. In the example scenario, all the fields are duplicated across multiple documents. When considering what relationships apply, it must be noted that there are no arrays; however duplication exists. The case with no arrays, but existing duplication is a many-to-one relationship. The scenario, therefore, should result in a string of many-to-one relationships.

```
{
    "_id" : ObjectId("583763acdf093fc9ca9cf93f"),
    "base" : 5,
    "sub" :
    {
        "duped" : "a",
        "alsoDuplicated" :
     {
            "z" : 24
        }
    }
}
```

Figure 42 Embedded Duplication Sample

When given the input of Figure 42, the system outputs the script in Figure 43. The script contains three tables and two many-to-one relationships. The first table is DeepDuplication, which represents the parent document. DeepDuplication contains the automatically generated DeepDuplicationId, the _id from MongoDB, and the copied field, base. DeepDuplication also contains subId to facilitate the many-to-one relationship with the table sub. The table sub is generated from the subdocument of the same name and contains the automatically generated id, subId, as well as the duped field, just as the subdocument had in the original document. The table, sub, also contains alsoDuplicatedId, which is used in reference to the alsoDuplicated table. The final table generated by the script is the alsoDuplicated table. The alsoDuplicated table is based on the deepest subdocument, contained in the alsoDuplicated field of the subdocument, sub. The alsoDuplicated table contains the automatically generated identifier, alsoDuplicatedId, and the z field, which was contained in the original subdocument. After table creation, the script generates the two many-to-one relationship constraints. First from DeepDuplication to sub, then from sub to alsoDuplicated.

```
CREATE TABLE DeepDuplication(
DeepDuplicationId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
base FLOAT(12) NOT NULL,
subId INT
)

CREATE TABLE sub(
subId INT IDENTITY(1,1) PRIMARY KEY,
duped VARCHAR(MAX) NOT NULL,
alsoDuplicatedId INT
)

CREATE TABLE alsoDuplicated(
alsoDuplicatedId INT IDENTITY(1,1) PRIMARY KEY,
z FLOAT(12) NOT NULL
)

--  Many-to-One Relationship
ALTER TABLE DeepDuplication
ADD CONSTRAINT FK_DeepDuplication_sub
FOREIGN KEY (subId)
REFERENCES sub(subId)

--  Many-to-One Relationship
ALTER TABLE sub
ADD CONSTRAINT FK_sub_alsoDuplicated
FOREIGN KEY (alsoDuplicatedId)
REFERENCES alsoDuplicated(alsoDuplicatedId)
```

Figure 43 Embedded Duplication Generation Script

After the system generates the tables, these tables can then be used by Entity Framework to generate the DTOs. The DTOs are then serialized to produce the structure in Table 9. The navigation properties have been removed to simplify the serialized DTO. The source document and the serializedDTO each have the fields base and sub. The field sub contains a subdocument with the fields duped and alsoDuplicated; the serialized DTO's subobject called sub contains the same properties. The subdocument alsoDuplicated contains a field z; the subobject alsoDuplicated in the serialized DTO also contains the field z. The structure of the serialized DTO is nearly the same as the original document.

MongoDB Document	Serialized DTO

<pre>{
    "_id" :
ObjectId("583763acdf093fc9ca9cf93f"),
    "base" : 5,
    "sub" :
    {
        "duped" : "a",
        "alsoDuplicated" :
     {
            "z" : 24
        }
    }
}</pre>
	<pre>{
	    "DeepDuplicationId": 0,
	    "C_id":
	"583763acdf093fc9ca9cf93f",
	    "base": 5.0,
	    "subId": 0,
	    "sub":
	    {
	        "subId": 0,
	        "duped": "a",
	        "alsoDuplicatedId": 0,
	        "alsoDuplicated":
	        {
	            "alsoDuplicatedId":
	0,
	            "z": 24.0
	        }
	    }
	}</pre>

Table 9 Embedded Duplication DTO Comparison

The system treats each relationship between layers independently, but it defaults to assuming duplication exists. There then comes a step to propagate duplication, where the system checks each document to confirm duplication. If a duplication does not exist, then relationships are traversed to propagate the lack of duplication upwards. Therefore, if the deepest subdocument is never duplicated, then it would force all of the containing documents to also no longer be able to be duplicated.

The example presented in Figure 42 has duplication throughout all subdocuments, as previously analyzed. However, if at the lowest layer, there is a subdocument that is not duplicated, it will break the duplications. Figure 44 contains the same structure as Figure 42, three layers, and the first two layers are duplicated across multiple documents.

However, the difference lies in the third layer, where NotQuiteDuplicated's subdocument, sub, contains a subdocument, notDuped, that is not duplicated. This makes the many-to-one relationships no longer valid.

```
{
    "_id" : ObjectId("58376334df093fc9ca9cf93d"),
    "base" : 5,
    "sub" :
    {
        "duped" : "a",
        "notDuped" :
        {
            "z" : 24
        }
    }
}
```

Figure 44 Multilayer Sample Document

The system sees this lack of duplication, and appropriately creates one-to-one relationships instead. Due to the methodology having the same implementation between the one-to-one and many-to-one relationship, the schema does not fully tell this story. However, there is a comment during script generation which indicates which kind of relationship was processed, as can be seen in Figure 45. The DeepDuplication collection was processed with two many-to-one relationships, while the NotQuiteDuplicated collection was processed with two one-to-one relationships.

```
CREATE TABLE NotQuiteDuplicated(
NotQuiteDuplicatedId INT IDENTITY(1,1) PRIMARY KEY,
_id VARCHAR(MAX) NOT NULL,
base FLOAT(12) NOT NULL,
subObjectNoDupeId INT
)

CREATE TABLE sub(
subId INT IDENTITY(1,1) PRIMARY KEY,
duped VARCHAR(MAX) NOT NULL,
notDupedId INT
)

CREATE TABLE notDuped(
notDupedId INT IDENTITY(1,1) PRIMARY KEY,
z FLOAT(12) NOT NULL
)

-- One-to-One Relationship
ALTER TABLE NotQuiteDuplicated
ADD CONSTRAINT FK_NotQuiteDuplicated_sub
FOREIGN KEY (subId)
REFERENCES sub(subId)

-- One-to-One Relationship
ALTER TABLE sub
ADD CONSTRAINT FK_sub_notDuped
FOREIGN KEY (notDupedId)
REFERENCES notDuped(notDupedId)
```

Figure 45 Multilayer Generation Script

After the generated schema is put through EntityFramework, the DTOs are serialized into JSON, and produce the results in Table 10. The figure omits the navigation properties that were automatically generated by Entity Framework. Comparing the documents to their comparable serialized objects, the fields are carried across the same. The parent has fields base and sub. The field sub contains a subdocument with the fields duped and notDuped. The field notDuped contains a subdocument with the field z. The original document structure can be generated from the table structure derived from the document collection.

MongoDB Document	Serialized DTO

<pre>{
    "_id" :
ObjectId("58376334df093fc9ca9cf93d"),
    "base" : 5,
    "sub" :
    {
        "duped" : "a",
        "notDuped" :
        {
            "z" : 24
        }
    }
}</pre>
	<pre>{
	    "NotQuiteDuplicatedId": 0,
	    "C_id":
	"58376334df093fc9ca9cf93d",
	    "base": 5.0,
	    "subId": 0
	    "sub":
	    {
	        "sub": 0,
	        "duped": "a",
	        "notDupedId": 0,
	        "notDuped":
	        {
	            "notDupedId": 0,
	            "z": 24.0
	        }
	    }
	}</pre>

Table 10 Multilayer DTO Comparison

Chapter 6

CONCLUSIONS

With the emerging popularity of NoSQL, it is becoming easier than ever to find methodologies for migrating into a NoSQL system. The topic receiving less attention, though, is how to migrate off NoSQL platforms. The described methodology seeks to make the gap smaller by easing the effort to the migration from a document database to a relational database system.

Chapter three contains a methodology which takes the implicit schema of a document database and automatically generates a relational database schema. The new schema will be able to receive the data of the document database when imported using other tools.

Chapter four covered an implementation of the methodology. The implementation used a slightly modified version of Variety, an open source tool for determining the schema of a MongoDB. The system then applied the methodology and generated a series of SQL scripts to generate the relational schema in a targeted relational database system.

Chapter five provided analysis of the generated schema. Entity Framework was used to generate C# classes from the generated relational database tables. These classes were serialized into JSON and compared to the documents in the document database. Upon

comparison, it was found that the serialized JSON strongly resembled the original document.

The provided methodology successfully generates a relational database schema that can contain the data which was stored in the original document database. Section 6.1 describes future work for the methodology. Section 6.2 analyzes the implementation of the methodology for some enhancements that will improve the system to be a more powerful tool. Section 6.3 provides concluding thoughts.

6.1 Future Work for the Methodology

In section 2.1.1, three ways were shown to generate a one-to-one relationship, either with a strong one-to-one relationship or a weak one-to-one relationship, which resembles a one-to-many relationship at the physical level. With the described methodology, the weak relationship was chosen to represent the one-to-one relationship in the generated schema. Choosing the strong relationship would result in the two generated tables in the one-to-one relationship to share a primary key. This could potentially lead to incorrect assumptions about the semantic relationships of the data, but it could eliminate redundancy in keys. How the one-to-one relationship is generated should be investigated further in future work. The future work would be to determine if the one-to-one relationship would be better represented by the current methodology's generation of two keys, or by a single shared primary key.

One flaw in the methodology is the scenario of multiple documents being named the same thing, despite being different subdocuments. This could lead to unrelated tables being merged in the current implementation. Therefore, future work on the implementation should take into account the location of the subdocument being converted. Utilizing the object path would avoid subdocuments on different object paths from being accidentally merged.

6.2 Future Work for the Implementation

There are additional improvements that could be made upon this implementation of this methodology. Most of the attention revolves around duplication detection. To detect duplications, the system currently loads all of document database into memory until it comes across a duplication, and then clears that cache. For very large document databases with rare or no duplications, this can lead to very inefficient memory use. Dynamically building queries to check for duplicates by querying against the document database could be leveraged to reduce memory use. At that point, an analysis would have to take place to determine what trade-offs in memory and input and output (I/O) would be acceptable.

Also, the current implemented system will result in false positives for duplication due to its checking of one field at a time. The system declares a document to be duplicated if all fields are duplicated at one point or another. It is possible for every field on a document to have duplicates, yet occur in different combinations, causing a false positive of

document duplication. An efficient way to check entire subdocuments would be necessary to make the duplication detection completely accurate. When the entire subdocument can be checked against every other subdocument, it becomes clear when a true duplication occurs.

A related problem which merits further investigation is the possibility that an entire chain of subdocuments is duplicated. A duplicated chain of subdocuments would be where a series of related subdocuments may be the same in all cases, and only related in a particular scenario. Figure 46 contains an example of this scenario. The current implementation will interpret a relationship between the root document and duplicatedSubDocument as a many-to-one, and then between duplicatedSubDocument and alwaysRelated as many-to-one, as well.

```
[
{'field':5, duplicatedSubDocument;{'subField':6,
'alwaysRelated':{'relField':9}}},
{'field':1, duplicatedSubDocument;{'subField':6,
'alwaysRelated':{'relField':9}}},
{'field':3, duplicatedSubDocument;{'subField':4,
'alwaysRelated':{'relField':3}}},
{'field':6, duplicatedSubDocument;{'subField':4,
'alwaysRelated':{'relField':3}}}
]
```

Figure 46 Duplicated Subdocument Chain

The true relationship is a many-to-one between the root document and duplicatedSubDocument. However, between duplicatedSubDocument and alwaysRelated is actually a one-to-one relationship. Note how any time the subField is

equal to 6, alwaysRelated's relField field has the value of 9. Any time the subField is equal to 4, relField is 2. This is actually a one-to-one relationship. The system should be able to detect this kind of relationship that is to a chain of subdocuments, rather than just one.

While checking for duplicates, a fuzzy matching could also be integrated. One of the motivations to migrate from a document database to relational database is to ensure consistency of data. Therefore, if data is inconsistent, the current implementation of equivalence matching may be insufficient. Adding a fuzzy matching algorithm would compensate for these scenarios of missing or incorrect data.

By a similar token, the data type conversion used in the implementation of the methodology is rudimentary, and could be greatly improved. At the end of the current implementation, most data is treated as either a float, date, or string. While correct, this loses the specificity of the data types, and could be corrected through use of algorithms used in data conversion.

The final thing which future work could expand on is the transferring of data. There exist tools which are able to transfer data using human intervention. With this novel technique for automatically generating relational database schema from a document database, it would be possible to generate a mapping between fields and columns. From this mapping, data transformation could take place to automatically migrate the data. This

would prove extremely useful in the scenarios that the methodology seeks to provide assistance in.

The described methodology could also be applied to a variety of areas beyond document databases. The methodology works off the principles of JSON, namely nested subobjects. XML also features nested subobjects, although they are less clear due to fields and objects being treated as new nodes. The methodology would work with flat files or strings of JSON or XML to generate relational database schemas.

6.3 Conclusions

During analysis of the methodology, it was demonstrated the ease at which a data structure is transformed from a document database, to a relational database, and back into a JSON structure. The resulting JSON structure is an accurate recreation of the document that served as input to the implementation.

Database systems are often observed as completely independent from each other; however, this work shows a consistent path, not only from document database to relational database, but also from a relational database to a document database. The described methodology automatically transforms a document collection into a relational database schema. During the analysis of the methodology, there was a demonstration of how an ORM is able to be utilized in the transformation of a relational database schema into the documents of a document database. Through the transformations, there were

only minor modifications to the structure. The original and final document structures were similar, and minor tweaks could be made to the ORM generation that would fix many of the flaws.

The common path for data migration today would require the collaboration of technical experts with domain experts to determine the final database schema. Then, analysis of the document database is necessary to discover what data formats exist. Due to the flexibility of the document database, data discovery could prove difficult. The technical experts would need to determine how to transform the data appropriately between systems.

Through use of the described methodology, a database schema was automatically generated which can contain the data of a previously existing document database. The generation was without any need for a domain expert, as the data was interpreted directly into the database schema. The automatic generation also handles all outliers from changes in the structure of the documents.

While the new schema will still need to be interpreted and understood, the automatic generation reduces the number of experts required to assist in the migration project. Due to the data being used to generate the database schema, the data migration will also be simplified. Once the database has been generated and populated, the schema can be reviewed and more easily modified, independently of the creation effort.

The methodology described by this thesis is a step towards allowing a more efficient translation of the implicit schema of a document database to the more traditional relational database schema. The methodology enables document database developers to more easily convert their system to utilize a relational database if the need arises.

REFERENCES

Print Publications:

[Boicea12]
Boicea, Alexandru, Florin Radulescu, and Laura Ioana Agapin. "MongoDB vs Oracle – database comparison", Third International Converence on Emerging Intelligent Data and Web Technologies, IEEE, 2012.

[Codd70]
Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM, ACM, June 1970.

[Date04]
Date, CJ. An Introduction to Database Systems. Pearson, Boston, 2004.

[Gomez16]
Gomez, Paolo, Rubby Casallas, and Claudia Roncancio. "Data Schema Does Matter, Even in NoSQL Systems". 2016 IEEE Tenth International Conference on Research Challenges in Information Science, IEEE, 2016.

[Goyal16]
Goyal, Akansha et al. "Cross Platform (RDBMS to NoSQL) Database Validation Tool using Bloom Filter", Fifth International Conference on Recent Trends in Information Technology, IEEE, 2016.

[Gyorodi15]
Gyorodi, Cornelia, et al. "A Comparative Study: MongoDB vs MySQL", 13th International Conference on Engineering of Modern Electric Systems, IEEE, 2015.

[Mohan13]
Mohan, C. "History Repeats Itself: Sensible and NonsenSQL Aspects of NoSQL Hoopla". Proceedings of the 16th International Conference on Extending Database Technology, ACM, 2013.

[Parker13]
Parker, Zachary, Scott Poe, and Susan Vrbsky. "Comparing NoSQL MongoDB to an SQL DB", Proceedings of the 51st ACM Southeast Conference, ACM, 2013.

[Zhao13]
Zhao, Gansen et al. "Modeling MongoDB with Relational Model", Fourth International Conference on Emerging Intelligent Data and Web Technologies, IEEE, 2013.

Electronic Sources:

[Couchbase16]
Couchbase. “Why NoSQL”. http://www.couchbase.com/nosql-resources/why-nosql, last accessed October 2, 2016.

[Dvorak16]
Dvořák, Tomáš, Eve Freeman, and James Cropcho. “Meet Variety, a Schema Analyzer for MongoDB”, https://github.com/variety/variety, August 10, 2016, last accessed November 21, 2016.

[Dziruko15]
Dziurko, Tomasz. “6 Things You Should Remember After GeeCON 2015”, http://tomaszdziurko.pl/2015/05/6-things-you-should-remember-after-geecon/, May 20th, 2015 , last accessed October 2, 2016.

[JSON16]
JSON. “Introducing JSON”. http://www.json.org/, last accessed November 21, 2016.

[Leanos16]
Leanos, Michael et. al. “MEANJS”. https://meanjs.org/, last accessed November 22, 2016.

[Mei13A]
Mei, Sarah. “Switching Data Stores: A Postmodern Comedy”. https://speakerdeck.com/sarahmei/switching-data-stores-a-postmodern-comedy, last revision October 18, 2013, last accessed October 2, 2016.

[Mei13B]
Mei, Sarah. “Why You Should Never Use MongoDB”. http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/, last revision November 11, 2013, last accessed October 2, 2016.

[Microsoft16]
Microsoft. “Entity Framework”. https://www.asp.net/entity-framework, last accessed November 27, 2016.

[MongoDB16A]
MongoDB. “Atomicity and Transactions”. https://docs.mongodb.com/v3.2/core/write-operations-atomicity/, last accessed November 23, 2016.

[MongoDB16B]
MongoDB. “JSON and BSON”. https://www.mongodb.com/json-and-bson, last accessed November 21, 2016.

[MongoDB16C]
MongoDB. “MongoDB Limits and Thresholds”. https://docs.mongodb.com/manual/reference/limits/, last accessed October 2, 2016.

[MongoDB16D]
MongoDB. “NOSQL Explained”. https://www.mongodb.com/nosql-explained, last accessed October 2, 2016.

[Mozilla17A]
Mozilla Developer Network. “Array.prototype.reduce()”. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce, last accessed February 23, 2017.

[Mozilla17B]
Mozilla Developer Network. “typeof”. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof, last accessed January 21, 2017.

[Node16]
Node. “About Node.js”. https://nodejs.org/en/about/, last accessed November 21, 2016.

[OpenSource.org17]
OpenSource.org. “The MIT License”. https://opensource.org/licenses/MIT, last accessed January 21, 2017.

[Quality Nonsense17]
Quality Nonsense. “Database Administration - ANSI SQL Standards and Guidelines”. http://www.whoishostingthis.com/resources/ansi-sql-standards/, last accessed January 2, 2017.

[RedHat16]
RedHat. “Hibernate”. http://hibernate.org/, last accessed December 24, 2016

[Sasaki15]
Sasaki, Bryce. “Graph Databases for Beginners: ACID vs. BASE Explained”. https://neo4j.com/blog/acid-vs-base-consistency-models-explained/, September 4, 2015, last accessed October 2, 2016

[W3Schools17]
W3Schools. “SQL PRIMARY KEY Constraint”. http://www.w3schools.com/sql/sql_primarykey.asp, last accessed January 1, 2017

[Young13]
Young, Greg. “Polyglot Data”. https://www.youtube.com/watch?v=GbM1ghLeweU, August 16, 2013, last accessed October 2, 2016.

APPENDIX A:

COMPLETE SOURCE CODE

```
#! /usr/bin/env node

'use strict';



// db.restaurants.find()

// mongo test --eval "var collection='restaurants',
persistResults=true,
resultsCollection='restaurantsKeys',outputFormat='json'"
"c:\mongodb\variety-cli-master\node_modules\variety\variety.js" --quiet



global.varietyResult = "";

var collectionName = "collectionName";



var cli = require('./variety-cli/lib/cli');

var promise = cli({

  stdin:  process.stdin,

  stdout: process.stdout,

  stderr: process.stderr,

  exitFn: process.exit,

  argv:   process.argv,

  process: process

});



var unprocessName = process.argv[2];

collectionName = unprocessName.split('/')[1];
```

```
promise.then(postPromise);



function postPromise(){
  console.log(global.varietyResult);
  var schemaAnalysis = JSON.parse(global.varietyResult);
  schemaAnalysis.forEach(processField)
  refineRelationships();
  var createScript = generateTableCreationSql();
  console.log(createScript);
  var alterScript = generateRelationshipSql();
  console.log(alterScript);

  /*var sql = require('mssql');

  sql.connect("mssql://app:password123@./Sample").then(
    function(){
      new sql.Request().batch(createScript + "\n\n" + alterScript);
    }
  )*/
}

var tables = [];
var relationships = [];

function processField(field){
  var tablesAndField = field._id.key.split(".");
  var fieldName = tablesAndField[tablesAndField.length - 1];
```

```
  var table = findTable(collectionName);


  if(tablesAndField.length == 1 && (typeof table == "undefined" ||
table == null)){

    table = {

          name : collectionName,

          fields: []

        };

    tables.push(table);

  } else {

    // Building tables in between

    for(var x = 0; x < tablesAndField.length - 1; x++){

      var tableName = tablesAndField[x];

      if(tablesAndField[x] == "XX"){

        tableName = tablesAndField[x-1] || collectionName;

        table = findTable(tableName);

      } else if(tablesAndField[x + 1] !== "XX"){

        table = findTable(tableName);

        if( typeof table == "undefined" || table == null ){

          table = {

            name : tableName,

            fields: [],

            hasDuplications: true

          };


          tables.push(table)


          generateRelationship("One", tablesAndField, x);
```

```
          if(tableName != collectionName){

            removeField(tableName, tablesAndField[x-1] ||
collectionName);

          }

        }

      }else{

        // Means array of objects, so its a one-to-many/many-to-many
between table before and table before that b/c the second table is an
array

        table = findTable(tableName);

        if( typeof table == "undefined" || table == null ){

          table = {

            name : tableName,

            fields: [],

            hasDuplications: true

          };



          tables.push(table);



          if(tableName != collectionName){

            removeField(tableName, tablesAndField[x-1] ||
collectionName);

          }

        }



        generateRelationship("Many", tablesAndField, x)

      }

    }

  }
```

```
  var dataType = determineDatatype(field.value.types);


  if(dataType !== null){ // Work-around for Arrays
    var fieldInformation = {
          name: fieldName,
          dataType: dataType, // dataType to be used in the SQL
          isNullable: field.percentContaining !== 100,
          hasDuplications: field.duplicationExists
        };


    // Assume there is duplication; as soon as a field exists that does
not, negate it.
    if(!field.duplicationExists){
      table.hasDuplications = false
    }


    table.fields.push(fieldInformation);
  }


/*
  Generating support structure


  Traverse varietyResult
    Split Key on .
    Check that each part but the last already exists in table Array
      If not exist, create
      Create Relationship
    Determine Data Type
```

```
    Check Duplications
    Add field to table
  */
}

function determineDatatype(types){
  /* passes in types array.
  Sample:
   "types" : {
         "Number" : 24608,
         "null" : 13
    }   */

    var dataTypes = Object.keys(types);
    var nullIndex = dataTypes.indexOf("null");
    if(nullIndex !== -1){
      dataTypes.splice(nullIndex, 1);
    }

    var finalDataType;
    if(dataTypes.length > 1){
      finalDataType = simplifyDataTypes(dataTypes);
    }else{
      finalDataType = dataTypes[0]
    }

    return chooseSqlDataType(finalDataType);
}
```

```
function simplifyDataTypes(types){
  if(types.indexOf("String") !== -1){
    return "String";
  } else if(types.indexOf("Float") !== -1){
    return "Float";
  }else if(types.indexOf("Number") !== -1){
    return "Number";
  } else {
    return "String";
  }
}

function chooseSqlDataType(type){
    switch(type){
      case "Number":
      case "Float":
        return "FLOAT(12)";
      case "String":
        return "VARCHAR(MAX)";
      case "Array":
        return "VARCHAR(MAX)";
      case "Boolean":
        return "BIT";
      case "Date":
        return "DATETIME";
      default:
        return "VARCHAR(MAX)";
```

```
    }
}


function generateRelationship(secondTableRelationshipType,
tablesAndField, curIndex){
  var firstTableName = collectionName;
  if( typeof tablesAndField[curIndex-1] !== "undefined"){
      firstTableName = tablesAndField[curIndex-1];
    }

  var rel = {
    firstTable: firstTableName,
    secondTable: tablesAndField[curIndex]
  }

  var sameRel = relationships.find(function(currentValue)
                        { return currentValue.firstTable ==
rel.firstTable
                                  && currentValue.secondTable ==
rel.secondTable; })
  if( typeof sameRel == "undefined" || sameRel == null){
    if(secondTableRelationshipType == "Many"){
      rel.relationshipType = RelationshipType.XToMany;
    } else {
      rel.relationshipType = RelationshipType.XToOne;
    }

    relationships.push(rel);
  }
```

```
}


var RelationshipType = {
  OneToOne: 1,
  OneToMany: 2,
  ManyToOne: 3,
  ManyToMany: 4,
  XToMany: 5,
  XToOne: 6
}


function removeField(fieldName, tableName){
  var table = table = findTable(tableName);
  var index = table.fields.findIndex(function(currentValue){ return
currentValue.name == fieldName; })


  if(index != -1){
    table.fields.splice(index, 1);
  }
}


var creationSql = "";
function generateTableCreationSql(){
/*
  Determine RelationshipType
  Traverse tables Array
    Create name + "Id IDENTITY(1,1) PRIMARY KEY"
    Iterate fields
```

```
      Name + " " + dataType

    Find all relationships with this table in first as One-to-Many OR
second in One-to-One or Many-to-One

      Add FK for each

*/

  creationSql ="";

    tables.forEach(createTable);


  return creationSql;

}


function refineRelationships(){

  /*

  relationships and tables are global...


  All relationshipType has   XToMany or XToOne: 6

  Use table.hasDuplications to determine which X it is

  */


  // TODO: Need to propogate duplicatedness down the object graph

  for(var x = 0; x < relationships.length; x++){

    /*

      Traverse, for each:

        If not a duplicate

          Propogate up the chain of relationships

    */


    var relationship = relationships[x];
```

```
    var destination = relationship.secondTable;

    var secondTable = findTable(destination);

    if(!secondTable.hasDuplications){

      propagateDuplication(relationship.firstTable);

    }

  }


  for(var x = 0; x < relationships.length; x++){

    var relationship = relationships[x];



    var tableName = relationship.secondTable;

    var table = findTable(tableName);



  if(table.hasDuplications && relationship.relationshipType ==
RelationshipType.XToMany){

      relationship.relationshipType = RelationshipType.ManyToMany;

    }else if(!table.hasDuplications && relationship.relationshipType ==
RelationshipType.XToMany){

      relationship.relationshipType = RelationshipType.OneToMany;

    }else if(!table.hasDuplications && relationship.relationshipType ==
RelationshipType.XToOne){

      relationship.relationshipType = RelationshipType.OneToOne;

    }else if(table.hasDuplications && relationship.relationshipType ==
RelationshipType.XToOne){

      relationship.relationshipType = RelationshipType.ManyToOne

    }else{

      throw "Invalid table+relationship";

    }

  }

}
```

```
function propagateDuplication(tableName){

  var firstTable = findTable(tableName);

  firstTable.hasDuplications = false;

  var relationshipsToModify = relationships.every(function(rel){ return
rel.secondTable === tableName })

  for(var y = 0; y < relationshipsToModify.length; y++){

    var relationship = relationshipsToModify[y];

    propagateDuplication(relationship.firstTable);

  }

}



function findTable(tableName){

  return tables.find(function(currentValue){ return currentValue.name
== tableName; });

}



function createTable(table){

  var create = "CREATE TABLE " + table.name;

  var pk = table.name + "Id" + " INT IDENTITY(1,1) PRIMARY KEY";



  var columns = "";

  for(var x = 0; x < table.fields.length; x++){

    var field = table.fields[x];

    columns = createColumn(columns, field)

  }



  var foreignKeys = createForeignKeys(table.name);
```

```
  creationSql = creationSql + "\n\n" + create + "(\n" + pk + ",\n" +
columns + (foreignKeys !== "" ? ",\n" + foreignKeys : "") + "\n)\n";

}



function createColumn(previousColumns, field){

  var starter = "";

  if(previousColumns !== ""){

    starter = previousColumns + ",\n";

  }



  return starter + field.name + " " + field.dataType +
(field.isNullable ? " NULL" : " NOT NULL");

}



function createForeignKeys(tableName){

  var foreignKeys = "";



  for(var x = 0; x < relationships.length; x++){

    var key = relationships[x];



    var starter = foreignKeys !== "" ? foreignKeys + ",\n" : "";



    if(key.firstTable == tableName && (key.relationshipType ===
RelationshipType.OneToOne || key.relationshipType ===
RelationshipType.ManyToOne)){

      foreignKeys = starter + key.secondTable + "Id INT";

    }else if(key.secondTable  == tableName && key.relationshipType ===
RelationshipType.OneToMany){

      foreignKeys = starter + key.firstTable + "Id INT";

    }
```

```
  }


  return foreignKeys;
}


/*
var table = {
    name : "",
    fields: [
      {
        name:"",
        dataType:"", // dataType to be used in the SQL
        isNullable:""
      }
    ]
  }


  var relationship = {
    firstTable: "",
    secondTable: "",
    relationshipType: "" ///// May be better to just go ahead and
designate source and target tables.....
      // If One-to-One or Many-to-One, FK goes on second table
      // If One-to-Many, FK goes on first table
      // If Many-to-Many, generate Bridge Table of name firstTable +
secondTable


      // Traverse the relationships for FK Constraint generation, also
  }*/
```

```
var alterRelationshipsSql = "";

function generateRelationshipSql(){

  relationships.forEach(generateRelationshipScript);



  return alterRelationshipsSql;

}



function generateRelationshipScript(relationship){

  var starter = alterRelationshipsSql === "" ? "\n\n" :
alterRelationshipsSql + "\n\n";



  if(relationship.relationshipType === RelationshipType.OneToOne ||
relationship.relationshipType === RelationshipType.ManyToOne){

    alterRelationshipsSql = starter + "-- " +
(relationship.relationshipType == RelationshipType.OneToOne ? "One-to-
One" : " Many-to-One")

    alterRelationshipsSql += " Relationship\n" +
createForeignKey(relationship.firstTable, relationship.secondTable);

  }else if(relationship.relationshipType ===
RelationshipType.OneToMany){

    alterRelationshipsSql = starter + "\n-- One-to-Many \n" +
createForeignKey(relationship.secondTable, relationship.firstTable);

  }else if(relationship.relationshipType ===
RelationshipType.ManyToMany){

    alterRelationshipsSql = starter + "\n-- Many-to-Many\n" +
generateManyToManyConstraint(relationship);

  }

}

/*

ALTER TABLE Orders

ADD CONSTRAINT fk_PerOrders
```

```
FOREIGN KEY (P_Id)

REFERENCES Persons(P_Id)

*/

function createForeignKey(startTable, destinationTable){

  var statement = "ALTER TABLE " + startTable +

    "\nADD CONSTRAINT FK_" + startTable + "_" + destinationTable +

    "\nFOREIGN KEY (" + destinationTable + "Id)\nREFERENCES " +

    destinationTable + "(" + destinationTable + "Id)";

  return statement;

}



function generateManyToManyConstraint(relationship){

  // Create Bridge Table

  var starter = "\n\n"



  var createTable = "CREATE TABLE " + relationship.firstTable +
relationship.secondTable;

  var firstColumn = relationship.firstTable + "Id INT REFERENCES " +
relationship.firstTable + "(" + relationship.firstTable + "Id)";

  var secondColumn = relationship.secondTable + "Id INT REFERENCES " +
relationship.secondTable + "(" + relationship.secondTable + "Id)";

  var compositeKey = "CONSTRAINT PK_" + relationship.firstTable +
relationship.secondTable + " PRIMARY KEY (" + relationship.firstTable +
"Id," + relationship.secondTable + "Id)";



  var createBridgeTableSql = starter + createTable + "(\n" +
firstColumn + ",\n" + secondColumn + ",\n" + compositeKey + "\n)";

  return createBridgeTableSql;

}
```

APPENDIX B:

MODIFIED VARIETY SOURCE CODE

```
/*

All modifications for Thesis by Jared Wheeler indicated by comment of
"Thesis Work"

*/



/* Variety: A MongoDB Schema Analyzer



This tool helps you get a sense of your application's schema, as well
as any

outliers to that schema. Particularly useful when you inherit a
codebase with

data dump and want to quickly learn how the data's structured. Also
useful for

finding rare keys.



Please see https://github.com/variety/variety for details.



Released by Maypop Inc, © 2012-2016, under the MIT License. */



(function () {

  'use strict'; // wraps everything for which we can use strict mode -
JC



  var log = function(message) {
```

```
    if(!__quiet) { // mongo shell param, coming from
https://github.com/mongodb/mongo/blob/5fc306543cd3ba2637e5cb0662cc375f3
6868b28/src/mongo/shell/dbshell.cpp#L624

      print(message);

    }

  };



  log('Variety: A MongoDB Schema Analyzer');

  log('Version 1.5.0, released 14 May 2015');



  var dbs = [];

  var emptyDbs = [];



  if (typeof slaveOk !== 'undefined') {

    if (slaveOk === true) {

      db.getMongo().setSlaveOk();

    }

  }



  var knownDatabases = db.adminCommand('listDatabases').databases;

  if(typeof knownDatabases !== 'undefined') { // not authorized user
receives error response (json) without databases key

    knownDatabases.forEach(function(d){

      if(db.getSisterDB(d.name).getCollectionNames().length > 0) {

        dbs.push(d.name);

      }

      if(db.getSisterDB(d.name).getCollectionNames().length === 0) {

        emptyDbs.push(d.name);

      }
```

```
    });


    if (emptyDbs.indexOf(db.getName()) !== -1) {
      throw 'The database specified ('+ db +') is empty.\n'+
           'Possible database options are: ' + dbs.join(', ') + '.';
    }


    if (dbs.indexOf(db.getName()) === -1) {
      throw 'The database specified ('+ db +') does not exist.\n'+
           'Possible database options are: ' + dbs.join(', ') + '.';
    }
  }


  var collNames = db.getCollectionNames().join(', ');
  if (typeof collection === 'undefined') {
    throw 'You have to supply a \'collection\' variable, à la --eval
\'var collection = "animals"\'.\n'+
          'Possible collection options for database specified: ' +
collNames + '.\n'+
          'Please see https://github.com/variety/variety for details.';
  }


  if (db.getCollection(collection).count() === 0) {
    throw 'The collection specified (' + collection + ') in the
database specified ('+ db +') does not exist or is empty.\n'+
          'Possible collection options for database specified: ' +
collNames + '.';
  }


  var readConfig = function(configProvider) {
```

```
    var config = {};

    var read = function(name, defaultValue) {

      var value = typeof configProvider[name] !== 'undefined' ?
configProvider[name] : defaultValue;

      config[name] = value;

      log('Using '+name+' of ' + tojson(value));

    };

    read('collection', null);

    read('query', {});

    read('limit',
db.getCollection(config.collection).find(config.query).count());

    read('maxDepth', 99);

    read('sort', {_id: -1});

    read('outputFormat', 'ascii');

    read('persistResults', false);

    read('resultsDatabase', 'varietyResults');

    read('resultsCollection', collection + 'Keys');

    read('resultsUser', null);

    read('resultsPass', null);

    read('logKeysContinuously', false);

    read('excludeSubkeys', []);

    read('arrayEscape', 'XX');



    //Translate excludeSubkeys to set like object... using an object
for compatibility...

    config.excludeSubkeys = config.excludeSubkeys.reduce(function
(result, item) { result[item+'.'] = true; return result; }, {});



    return config;

  };
```

```
  var config = readConfig(this);


  var PluginsClass = function(context) {

    var parsePath = function(val) { return val.slice(-3) !== '.js' ?
val + '.js' : val;};

    var parseConfig = function(val) {

      var config = {};

      val.split('&').reduce(function(acc, val) {

        var parts = val.split('=');

        acc[parts[0]] = parts[1];

        return acc;

      }, config);

      return config;

    };


    if(typeof context.plugins !== 'undefined') {

      this.plugins = context.plugins.split(',')

      .map(function(path){return path.trim();})

      .map(function(definition){

        var path = parsePath(definition.split('|')[0]);

        var config = parseConfig(definition.split('|')[1] || '');

        context.module = context.module || {};

        load(path);

        var plugin = context.module.exports;

        plugin.path = path;

        if(typeof plugin.init === 'function') {

          plugin.init(config);
```

```
        }

        return plugin;

      }, this);

    } else {

      this.plugins = [];

    }



    this.execute = function(methodName) {

      var args = Array.prototype.slice.call(arguments, 1);

      var applicablePlugins =
this.plugins.filter(function(plugin){return typeof plugin[methodName]
=== 'function';});

      return applicablePlugins.map(function(plugin) {

        return plugin[methodName].apply(plugin, args);

      });

    };



    log('Using plugins of ' +
tojson(this.plugins.map(function(plugin){return plugin.path;})));

  };



  var $plugins = new PluginsClass(this);

  $plugins.execute('onConfig', config);



  var varietyTypeOf = function(thing) {

    if (typeof thing === 'undefined') { throw 'varietyTypeOf() requires
an argument'; }



    if(typeof thing === "number" && thing.toString().indexOf('.') !== -
1){
```

```
      return 'Float'
    } else if (typeof thing !== 'object') {
    // the messiness below capitalizes the first letter, so the output
matches
    // the other return values below. -JC
      var typeofThing = typeof thing; // edgecase of JSHint's
"singleGroups"
      return typeofThing[0].toUpperCase() + typeofThing.slice(1);
    } else {
      if (thing && thing.constructor === Array) {
        return 'Array';
      } else if (thing === null) {
        return 'null';
      } else if (thing instanceof Date) {
        return 'Date';
      } else if(thing instanceof NumberLong) {
        return 'NumberLong';
      } else if (thing instanceof ObjectId) {
        return 'ObjectId';
      } else if (thing instanceof BinData) {
        var binDataTypes = {};
        binDataTypes[0x00] = 'generic';
        binDataTypes[0x01] = 'function';
        binDataTypes[0x02] = 'old';
        binDataTypes[0x03] = 'UUID';
        binDataTypes[0x05] = 'MD5';
        binDataTypes[0x80] = 'user';
        return 'BinData-' + binDataTypes[thing.subtype()];
      } else {
```

```
        return 'Object';
      }
    }
  };


  //flattens object keys to 1D. i.e. {'key1':1,{'key2':{'key3':2}}}
becomes {'key1':1,'key2.key3':2}

  //we assume no '.' characters in the keys, which is an OK assumption
for MongoDB

  var serializeDoc = function(doc, maxDepth, excludeSubkeys) {

    var result = {};



    //determining if an object is a Hash vs Array vs something else is
hard

    //returns true, if object in argument may have nested objects and
makes sense to analyse its content

    function isHash(v) {

      var isArray = Array.isArray(v);

      var isObject = typeof v === 'object';

      var specialObject = v instanceof Date ||

                          v instanceof ObjectId ||

                          v instanceof BinData ||

                          v instanceof NumberLong;

      return !specialObject && (isArray || isObject);

    }



    var arrayRegex = new RegExp('\\.' + config.arrayEscape + '\\d+' +
config.arrayEscape + '\\.', 'g');



    function serialize(document, parentKey, maxDepth) {
```

```
      if(Object.prototype.hasOwnProperty.call(excludeSubkeys,
parentKey.replace(arrayRegex, '.')))

        return;

      for(var key in document) {

        //skip over inherited properties such as string, length, etch

        if(!document.hasOwnProperty(key)) {

          continue;

        }

        var value = document[key];

        if(Array.isArray(document))

          key = config.arrayEscape + key + config.arrayEscape;
//translate unnamed object key from {_parent_name_}.{_index_} to
{_parent_name_}.arrayEscape{_index_}arrayEscape.

        result[parentKey+key] = value;

        //it's an object, recurse...only if we haven't reached max
depth

        if(isHash(value) && maxDepth > 1) {

          serialize(value, parentKey+key+'.', maxDepth-1);

        }

      }

    }

    serialize(doc, '', maxDepth);

    return result;

  };



  // convert document to key-value map, where value is always an array
with types as plain strings

  var values = {}; // Thesis Work

  var duplicationTracking = {}; // Thesis Work

  var analyseDocument = function(document) {
```

```
    var result = {};

    var arrayRegex = new RegExp('\\.' + config.arrayEscape + '\\d+' +
config.arrayEscape, 'g');

    for (var key in document) {

      var value = document[key];

      key = key.replace(arrayRegex, '.' + config.arrayEscape);

      if(typeof result[key] === 'undefined') {

        result[key] = {};

      }

      var type = varietyTypeOf(value);

      result[key][type] = true;


      // Thesis Work Start

      if(typeof values[key] === 'undefined') {

        values[key] = {};

        values[key].values = [];

        duplicationTracking[key] = false; // initialize to False

      }


      if(duplicationTracking[key] === false){

        if(values[key].values.indexOf(value.toString()) >= 0){

          duplicationTracking[key] = true; // Will only get set to true
under this one circumstance

          log(key + " has a duplicate detected of " + value);

          values[key].values = [];

        }else{

          values[key].values.push(value.toString())

          log(key + " has NO duplicate detected of " + value);

        }
```

```
      }
      // Thesis Work End
    }
    return result;
  };

  var mergeDocument = function(docResult, interimResults) {
    for (var key in docResult) {
      if(key in interimResults) {
        var existing = interimResults[key];
        //log(key + ": " + JSON.stringify(existing));
        for(var type in docResult[key]) {
          if (type in existing.types) {
            existing.types[type] = existing.types[type] + 1;
          } else {
            existing.types[type] = 1;
            if (config.logKeysContinuously) {
              log('Found new key type "' + key + '" type "' + type +
'"');
            }
          }
        }
        existing.totalOccurrences = existing.totalOccurrences + 1;
      } else {
        var types = {};
        for (var newType in docResult[key]) {
          types[newType] = 1;
          if (config.logKeysContinuously) {
```

```
            log('Found new key type "' + key + '" type "' + newType +
'"');
          }
        }
        interimResults[key] = {'types': types,'totalOccurrences':1};
      }
    }
  };

  var convertResults = function(interimResults, documentsCount) {
    var getKeys = function(obj) {
      var keys = {};
      for(var key in obj) {
        if(key !== "duplicationExists"){
          keys[key] = obj[key];
        }
      }
      return keys;
    //return keys.sort();
    };
    var varietyResults = [];
    //now convert the interimResults into the proper format
    for(var key in interimResults) {
      var entry = interimResults[key];
      varietyResults.push({
        '_id': {'key':key},
        'value': {'types':getKeys(entry.types)},
        'totalOccurrences': entry.totalOccurrences,
```

```
        'percentContaining': entry.totalOccurrences * 100 /
documentsCount,

        'duplicationExists': duplicationTracking[key] // Thesis Work

      });

      log(key + " has a duplicationTracking value of " +
duplicationTracking[key])

    }

    return varietyResults;

  };



  // Merge the keys and types of current object into accumulator object

  var reduceDocuments = function(accumulator, object) {

    var docResult = analyseDocument(serializeDoc(object,
config.maxDepth, config.excludeSubkeys));

    mergeDocument(docResult, accumulator);

    return accumulator;

  };



  // We throw away keys which end in an array index, since they are not
useful

  // for our analysis. (We still keep the key of their parent array,
though.) -JC

  var arrayRegex = new RegExp('\\.' + config.arrayEscape + '$', 'g');

  var filter = function(item) {

    return !item._id.key.match(arrayRegex);

  };



// sort desc by totalOccurrences or by key asc if occurrences equal

  var comparator = function(a, b) {

    var countsDiff = b.totalOccurrences - a.totalOccurrences;
```

```
    return countsDiff !== 0 ? countsDiff :
a._id.key.localeCompare(b._id.key);

  };



  // extend standard MongoDB cursor of reduce method - call forEach and
combine the results

  DBQuery.prototype.reduce = function(callback, initialValue) {

    var result = initialValue;

    this.forEach(function(obj){

      result = callback(result, obj);

    });

    return result;

  };



  var cursor =
db.getCollection(config.collection).find(config.query).sort(config.sort
).limit(config.limit);

  var interimResults = cursor.reduce(reduceDocuments, {});

  var varietyResults = convertResults(interimResults, cursor.size())

  .filter(filter)

  .sort(comparator);



  if(config.persistResults) {

    var resultsDB;

    var resultsCollectionName = config.resultsCollection;



    if (config.resultsDatabase.indexOf('/') === -1) {

    // Local database; don't reconnect

      resultsDB = db.getMongo().getDB(config.resultsDatabase);

    } else {
```

```
    // Remote database, establish new connection
      resultsDB = connect(config.resultsDatabase);
    }

    if (config.resultsUser !== null && config.resultsPass !== null) {
      resultsDB.auth(config.resultsUser, config.resultsPass);
    }

    // replace results collection
    log('replacing results collection: '+ resultsCollectionName);
    resultsDB.getCollection(resultsCollectionName).drop();
resultsDB.getCollection(resultsCollectionName).insert(varietyResults);
  }

  var createAsciiTable = function(results) {
    var headers = ['key', 'types', 'occurrences', 'percents'];
    // return the number of decimal places or 1, if the number is int
(1.23=>2, 100=>1, 0.1415=>4)
    var significantDigits = function(value) {
      var res = value.toString().match(/^[0-9]+\.([0-9]+)$/);
      return res !== null ? res[1].length : 1;
    };

    var maxDigits = varietyResults.map(function(value){return
significantDigits(value.percentContaining);}).reduce(function(acc,val){
return acc>val?acc:val;});

    var rows = results.map(function(row) {
      var types = [];
```

```
      var typeKeys = Object.keys(row.value.types);

      if (typeKeys.length > 1) {

        for (var type in row.value.types) {

          var typestring = type + ' (' + row.value.types[type] + ')';

          types.push(typestring);

        }

      } else {

        types = typeKeys;

      }



      return [row._id.key, types, row.totalOccurrences,
row.percentContaining.toFixed(Math.min(maxDigits, 20))];

    });

    var table = [headers, headers.map(function(){return
'';})].concat(rows);

    var colMaxWidth = function(arr, index) {return Math.max.apply(null,
arr.map(function(row){return row[index].toString().length;}));};

    var pad = function(width, string, symbol) { return width <=
string.length ? string : pad(width, isNaN(string) ? string + symbol :
symbol + string, symbol); };

    table = table.map(function(row, ri){

      return '| ' + row.map(function(cell, i) {return
pad(colMaxWidth(table, i), cell.toString(), ri === 1 ? '-' : '
');}).join(' | ') + ' |';

    });

    var border = '+' + pad(table[0].length - 2, '', '-') + '+';

    return [border].concat(table).concat(border).join('\n');

  };



  /*var pluginsOutput = $plugins.execute('formatResults',
varietyResults);

  if (pluginsOutput.length > 0) {
```

```
    pluginsOutput.forEach(function(i){print(i);});

  } else if(config.outputFormat === 'json') {

    printjson(varietyResults); // valid formatted json output,
compressed variant is printjsononeline()

  } else {

    print(createAsciiTable(varietyResults)); // output nice ascii table
with results

  }*/

  printjson(varietyResults);



}.bind(this)()); // end strict mode
```

VITA

Jared Wheeler is a student at the University of North Florida, pursuing a joint Bachelors-Masters degree, which he expects to earn in the Spring of 2017. Dr. Behrooz Seyed-Abbassi of the University of North Florida served as Jared's thesis advisor. Jared is working as a Software Engineer II at Beeline, and has been employed there since joining as an intern in the Fall of 2012.

Jared has ongoing interests in artificial intelligence, game design, and semantics. He works extensively with C#, and has experience with JavaScript. Jared's academic work includes Java and C, with some projects in Prolog and Ruby.