Renormalization of NoSQL Database Schemas

Michael J. Mior Kenneth Salem

Abstract

NoSQL applications often use denormalized databases in order to meet performance goals, but this introduces complications. In particular, application evolution may demand changes in the underlying database schema, which may in turn require further application revisions. The NoSQL DBMS itself can do little to aid in this process, as it has no understanding of application-level denormalization.

In this paper, we describe a procedure for reconstructing a normalized conceptual schema from a denormalized NoSQL database. Exposing the conceptual schema provides application developers with information that can be used to guide application and database evolution. The procedure's input includes functional and inclusion dependencies, which may be mined from the NoSQL database. We illustrate the effectiveness of this procedure using several application case studies.

As discussed in the previous chapter, although NoSQL systems may not require applications to define rigid schemas, application developers must still decide how to store information in the database. These choices can have a significant impact on application performance as well as the readability of application code [14]. For example, consider an application using HBase to track requests to an on-line service. To store records of requests in an HBase table, the application must decide how to represent the requests. Should there be a record for each request, or perhaps one record for all the requests from a single client? What column families will be present in the table, and what will they represent? The same requests may be stored in multiple tables since the structure of tables determines which queries can be asked. The choice of data representation depends on how the application expects to use the table, i.e., what kinds of queries and updates it needs to perform. These kinds of decisions are automated with NoSE, but many existing applications have manually designed schemas. In this case, since the NoSQL system itself is unaware of any schema design decisions, it can provide little to no help in understanding what is being represented in the database.

In our on-line service example, request information might be stored twice, once grouped and keyed by the customer that submitted the request, and a second time keyed by the request subject or the request time. If the application updates a request, or changes the information it tracks for each request, these changes should be reflected in both locations. This denormalization (duplication of data) is done by the application developer. The database system is unaware of this denormalization and unable to help manage updates and queries over this denormalized data. We aim to recover explicit knowledge of application-level denormalization through a task we refer to as *schema renormalization*. This chapter addresses the schema renormalization problem through the following technical contributions:

- We present a complete semi-automatic technique for extracting a normalized conceptual schema from an existing denormalized NoSQL database. Our technique works with different types of NoSQL systems (e.g., Cassandra, MongoDB) through a common unifying relational representation of the physical structures in the NoSQL database. It produces a normalized conceptual schema for the database, such as the one represented graphically an entity-relationship diagram, in Figure 1. In addition, it produces a *mapping* from each NoSQL physical structure to the conceptual model. Connecting the physical and conceptual schemas this way increases their utility, as discussed in Section 6.

- We develop an automatic normalization algorithm (Section 4), which forms the core of our full semi-automatic approach. This algorithm uses both functional and inclusion dependencies to extract the conceptual model from the NoSQL system's physical structures. Our algorithm ensures that the resulting model is in *interaction-free inclusion dependency normal form*, indicating the redundancy implied by the input dependencies (both functional and inclusion) has been removed from the schema. To the best of our knowledge, this is the first normalization algorithm which does this.

- Our normalization algorithm requires functional and inclusion dependencies as input. These may be provided by a knowledgeable user, or may be mined from an instance of the NoSQL databases. Our third contribution is a technique for adapting existing relational dependency mining techniques to provide the dependencies required by the normalization algorithm (Section 5).

- Finally, in Section 7, we present a series of case studies which show the full schema renormalization process in action for several NoSQL applications. We use these case studies to highlight both the advantages and the limitations of our approach to renormalization.

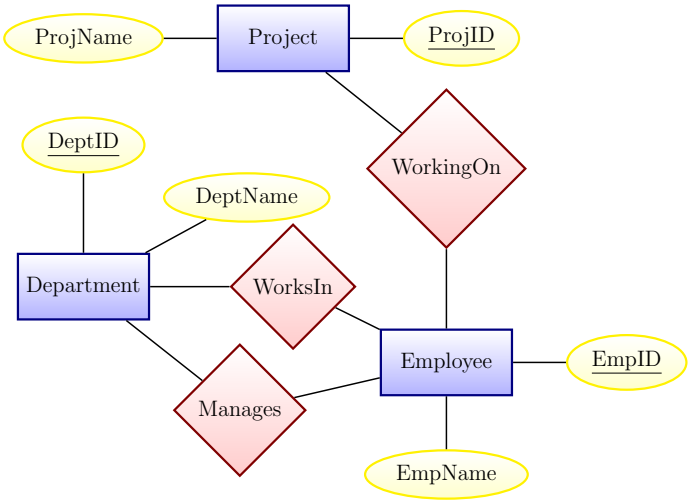

Figure 1: Schema example after renormalization

The conceptual data model that our algorithm produces can serve as a simple reference, or specification, of the information that has been denormalized across the workload-tuned physical database structures. We view this model as a key component in a broader methodology for schema management for NoSQL applications. Current processes for managing schema evolution in NoSQL datastores are entirely manual and error prone. If the application workload changes, or if new types of data are added to the database, a developer must evaluate how the physical schema must evolve to support the change. This process must be repeated each time the application evolves, with the potential for errors each time.

We would like to support an alternative methodology for schema evolution, which is illustrated in Figure 2. The first step is construction of a normalized conceptual data model over the denormalized schema. This is the problem we address in this thesis. This conceptual model enables several new use cases. Through the knowledge of the denormalization present in the existing physical structures, we can determine efficient plans for executing ad-hoc queries over the denormalized data. In addition, the application developer then can "lift" existing applications by describing the application's existing queries and updates against the conceptual model. Instead of directly changing the physical schema, application developers can then evolve the application at the level of the conceptual model, e.g., by adding additional data, or by adding or modifying (conceptual) queries. Once the application has been evolved at the conceptual level, existing schema design tools and techniques, such as NoSE and NoAM [3], can then be used to generate a new, workload-aware, denormalized physical database design for the target NoSQL system.

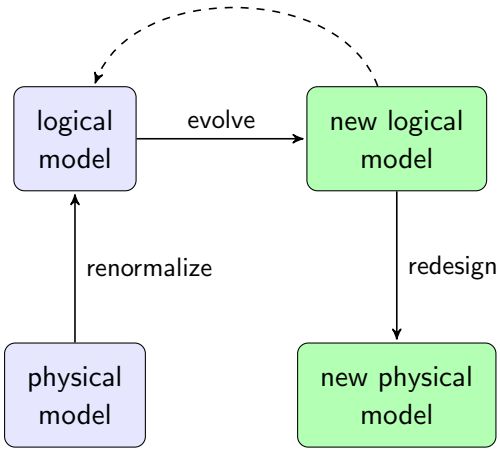

Figure 2: NoSQL schema evolution lifecycle

1 Renormalization Overview

We renormalize NoSQL databases using a three step process. The first step is to produce a *generic physical schema* that describes the physical structures that are present in the NoSQL database. This step may be performed manually, although some tools exist to aid in automation, which we discuss later. The generic physical schema serves to abstract differences among the database models of different types of NoSQL systems. For example, HBase uses tables with one or more column families, while MongoDB stores collections of JSON documents. The generic physical schema hides these differences, providing a uniform way to represent the physical structures that are present in the NoSQL store. It does not capture all of the characteristics of these structures. In particular, it does not capture how they can be used by applications, and it does not capture features of the structures that affect performance. Rather, it focuses on describing the information that is present in these structures, which is what is needed for renormalization. We describe the generic physical model in more detail in Section 2, and illustrate how it can be produced for different types of NoSQL systems.

The second step in the renormalization process is to identify dependencies among the attributes of the generic physical model. The required dependencies can be provided by a user with understanding of the NoSQL system's application domain [20] or automatically using existing dependency mining techniques, which we explore in Section 5. We discuss the required dependencies further in Section 3.

The final step in the renormalization process is to normalize the generic physical schema using the dependencies, resulting in a logical schema such as the one represented (as an ER diagram) in Figure 1. This step is automated, using the procedure described in Section 4. Our algorithm ensures that the normalized schema is in *inclusion dependency normal form (IDNF)* which informally means that redundancy in the physical schema captured by the provided functional and inclusion dependencies is removed.

Although we do not discuss this further in this thesis, it is also possible to apply this three-step methodology iteratively, to incrementally renormalize a database. In particular, one can start with a *partial* physical schema (e.g. a single table), renormalize it, and then gradually add to the schema and renormalize until the full physical schema has been renormalized.

2 The Generic Physical Schema

The first step in the renormalization process is to describe the NoSQL database using a generic schema. The schemas we use are relational. Specifically, a generic physical schema consists of a set of *relation schemas*. Each relation schema describes a physical structure in the underlying NoSQL database. A relation schema, in turn, consists of a unique relation name plus a fixed set of attribute names. Attribute names are unique within each relation schema.

The procedure for doing this depends on the type of NoSQL database that is being normalized. Here, we illustrate the process using examples based on three different types of systems: Cassandra, HBase, and MongoDB. Our examples are based on RUBiS, an online auction application which we describe in more detail later, in Section 7.1.

Cassandra: NoSQL systems differ in the amount of schema information that they understand. In Cassandra, data is stored in tables, which applications can define using CQL, an SQL-like language. CQL includes a `CREATE TABLE` statement, which allows the application to define the structure of a table. Figure 3 shows an example of a CQL definition of a single table from the RUBiS database. This table records the IDs of bids for each item under auction. Information about the bids, such as the bid date, is denormalized into this table so that an application can retrieve it without having to perform joins, which Cassandra does not support.

If the NoSQL database includes a well-defined schema, as in this example, then describing the physical schema required for renormalization is a trivial task. Figure 3 also

CQL

```
CREATE TABLE ItemBids(itemID uuid, bid decimal,
bidID uuid, quantity int, date timestamp,
PRIMARY KEY(itemID,bid,bidID));
```

Generic schema

```
ItemBids(itemID, bid, bidID, quantity, date)
```

Figure 3: A CQL **ItemBids** table, and corresponding schema

2315	bids:25,b9734 16,2016-05-02	bids:24,b3267 6,2016-05-02	bids:22,b9907 8,2016-05-01
2416	bids:65,b7633 1,2016-04-09	bids:60,b9028 ,2016-04-01	

Figure 4: An **ItemBids** table in HBase

shows the generic relation schema for the CQL **ItemBids** table. The generic schema simply identifies the attributes that are present in the table, and gives names to both the attributes and the table itself. In the case of Cassandra, these names can be taken directly from the CQL table definition. The `PRIMARY KEY` declaration in the CQL table definition also provides information about functional dependencies among the tables attributes. We defer the further discussion of these dependencies to Section 3.

HBase: Like Cassandra, HBase stores data in tables. Each table contains one or more column families. However, HBase understands only table names and the names of the tables' column families. Individual columns in each column family are not fixed. Different rows in the same table may have different columns. Figure 4 shows two rows from an HBase table that stores the same information (about bids for each item) that a Cassandra application would store in the table from Figure 3. The table includes one row per item, and a single column family called **bids**. In each row, there is a column for each bid for that row's item. Column names are composite values representing the bid amount and a bid identifier (a reference to a row in another HBase table). Cells hold composite values identifying the bid quantity and bid date.

The HBase **ItemBids** table can be modeled by the same generic schema that was shown in Figure 3. In this case, each row in the HBase table results in *multiple* rows in the

```
{ _id:  2315, bids: [
  { _id: b9734, amount: 25,
     quantity: 16, date: "2016-05-02"},
  { _id: b3267, amount: 24,
     quantity: 6, date: "2016-05-02"},
  { _id: b9907, amount: 22,
     quantity: 8, date: "2016-05-01"}
]}
{ _id:  2416, bids: [
  { _id: b7633, amount: 65,
     quantity: 1, date: "2016-04-09"},
  { _id: b9028, amount: 60, date: "2016-04-01"}
]}
```

Figure 5: An **ItemBids** collection in MongoDB

generic relation — one row per bid. To identify this model, the user must understand that column names are composites of bid values and bid identifiers, and similarly that the cell values are composites. The user must also understand that row keys are item identifiers. This interpretation is commonly imposed on the data when it is read from the HBase table by an application. Thus, a user with knowledge of the application can identify attributes either directly from the database or through their application knowledge.

MongoDB: Unlike Cassandra and HBase, MongoDB stores data in collections of documents. Each document in a collection is a JSON object containing at minimum a primary key. The only metadata available from MongoDB is the names of these collections. While each document is permitted to contain arbitrary JSON data, in practice, documents within the same collection have some common structure. Figure 5 shows how the same information that is recorded in the HBase table from Figure 4 might be represented in a MongoDB document collection, **ItemBids**. Each document contains an ID as well as an array of bids for the item. The ID for each bid references another collection. The **ItemBids** collection could be modeled using the same generic relation schema that was used to model the **ItemBids** table for HBase and Cassandra.

In general, we anticipate that the definition of a generic physical schema for an application will require user involvement. However, there are tools that may assist with this process. For example, several authors have proposed methods for extracting a schema from JSON records in a document store, which could be applied to extract the generic physical schema required for renormalization [16, 17, 27]. These methods generate nested schemas,

but nested properties can be flattened by concatenating their names. Similarly, arrays can be flattened by including multiple rows for each document, as we have done in this example.

3 Dependency Input

The second step of the renormalization process is to identify dependencies among the attributes in the generic physical schema. Our normalization algorithm is able to use two types of dependencies: functional dependencies and inclusion dependencies. These two forms of dependencies are easy to express and are commonly used in database design [19].

These dependencies can be input manually, by a user who is familiar with the application and its database. Alternatively, dependencies can be mined from an instance of the underlying NoSQL database. In this section, we describe the types of dependencies that our normalization algorithm expects. We defer discussion of dependency mining to Section 5.

Functional dependencies (FDs) are of the form $R : A \rightarrow B$, where R is a relation from the physical schema and A and B are sets of attributes from R. For example, for the **ItemBids** relation described in Section 2, the user might identify the following functional dependencies:

$$\begin{aligned} \text{itemID}, \text{bid}, \text{bidID} &\rightarrow \text{quantity}, \text{date} \\ \text{bidID} &\rightarrow \text{itemID}, \text{bid}. \end{aligned}$$

The first may be identified because **itemID**, **bid**, and **bidID** together form a row key for the physical relation. The latter may be identified based on knowledge of the application domain. We expect the functional dependencies provided as input to our algorithm are given in the order they should be processed. That is, the schema will be normalized starting with the first dependency in the list.

Inclusion dependencies (INDs) are of the form $R(A) \subseteq S(B)$ where R and S are physical relations, A is a set of attributes in R and B is a set of attributes in S. The dependency states that for any tuple in R, there exists a tuple in S where the values of attributes in B match the values of the attributes in A for the tuple in R. To represent both the inclusion dependencies $R(A) \subseteq S(B)$ and $S(B) \subseteq R(A)$, we use the shorthand $R(A) = S(B)$. Inclusion dependencies are useful to determine when an application has duplicated attributes across multiple physical structures.

For input to our algorithm, we require that all INDs are superkey-based. That is, for an IND $R(A) \subseteq S(B)$, B must be a superkey of S. We do not believe that this is a significant restriction since we intend for inclusion dependencies to be used to indicate foreign key relationships which exist in the denormalized data. Indeed, Mannila and Räihä [19] have previously argued that only key-based dependencies are relevant to logical design.

4 Normalization Algorithm

Levene and Vincent [18] define a normal form for database relations involving functional and inclusion dependencies referred to as inclusion dependency normal form (IDNF). They have shown that normalizing according to IDNF removes redundancy from a database design implied by the set of dependencies. However, one of the necessary conditions for this normal form is that the set of inclusion dependencies is non-circular. A set of inclusion dependencies $\mathrm{I}_1 : R_1(A_1) \subseteq \ldots \subseteq \mathrm{I}_n : R_n(A_n)$ is circular if $R_1 = R_n$. This excludes useful schemas which express constraints such as one-to-one foreign key integrity. For example, for the relations $R(\underline{A}, B)$ and $S(\underline{B}, C)$ we can think of the circular inclusion dependencies $R(A) = S(B)$ as expressing a one-to-one foreign key between $R(A)$ and $S(B)$.

Levene and Vincent also propose an extension to IDNF, termed *interaction-free inclusion dependency normal form* which allows such circularities. The goal of our normalization algorithm is to produce a schema that is in interaction-free IDNF. This normal form avoids redundancy implied by functional and inclusion dependencies while still allowing the expression of useful information such as foreign keys. We provide more details on this normal form in Section 4.5. As we show in Section 7, this produces useful logical models for several real-world examples.

Figure 6 provides an overview of our normalization algorithm, which consists of four stages. In the reminder of this section, we discuss the normalization algorithm in more detail. We will make use of a running example based on the simple generic (denormalized) physical schema and dependencies shown in Figure 7.

Our normalization algorithm first applies dependency inference rules, as we discuss in Section 4.1. Second, the `BCNFDecompose` algorithm implements BCNF decomposition. This removes any redundancy according to the set of FDs. Next, the `Fold` algorithm removes redundant attributes and relations according to the set of INDs. Finally, `BreakCycles` breaks any inclusion dependency cycles which are not proper circular, to ensure that the resulting schema is in interaction-free IDNF.

Data: A set of relations $\mathbf{R}$, FDs $\mathbf{F}$, and INDs $\mathbf{I}$
Result: A normalized set of relations $\mathbf{R}'''$
begin

```
// Perform dependency inference
F', I+ ← Expand(F, I)
// Normalize according to BCNF
R', I+' ← BCNFDecompose(R,F',I+)
// Remove redundant attributes and relations
R'', I+'' ← Fold(R', F', I+')
// Break remaining circular INDs
R''', I+''' ← BreakCycles(R'', I+'')
```

Figure 6: Algorithm for normalization to interaction-free IDNF

4.1 Dependency Inference

To minimize the effort required to provide input needed to create a useful normalized schema, we aim to infer dependencies whenever possible. Armstrong [2] provides a well-known set of axioms which can be used to infer FDs from those provided as input. Similarly, Mitchell [22] presents a similar set of inference rules for INDs.

Mitchell further presents a set of inference rules for joint application to a set of FDs and INDs. We adopt Mitchell's pullback and collection rules to infer new functional dependencies for inclusion dependencies and vice versa. As an example of the pullback rule, consider the following dependencies for our example in Figure 7:

$$\begin{gathered}\text{Employees} : \text{EmpID} \rightarrow \text{EmpName} \\ \text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) \subseteq \text{Employees}\,(\text{EmpID}, \text{EmpName})\,.\end{gathered}$$

In this case, we are able to infer an additional functional dependency:

$$\text{EmpProjects} : \text{EmpID} \rightarrow \text{EmpName}.$$

This case is equivalent to propagating primary keys of different logical entities (in this case, employees) across different relations.

Physical Schema

$$\begin{aligned}\text{EmpProjects}(&\underline{\text{EmpID}}, \text{EmpName}, \underline{\text{ProjID}}, \text{ProjName})\\ \text{Employees}(&\underline{\text{EmpID}}, \text{EmpName}, \text{DeptID}, \text{DeptName})\\ \text{Managers}(&\underline{\text{DeptID}}, \text{EmpID})\end{aligned}$$

Functional Dependencies

$$\begin{aligned}\text{Employees} : \text{EmpID} &\rightarrow \text{EmpName}, \text{DeptID}\\ \text{Employees} : \text{DeptID} &\rightarrow \text{DeptName}\\ \text{EmpProjects} : \text{ProjID} &\rightarrow \text{ProjName}\\ \text{Managers} : \text{DeptID} &\rightarrow \text{EmpID}\end{aligned}$$

Inclusion Dependencies

$$\begin{aligned}\text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots)\\ \text{Managers}\,(\text{EmpID}) &\subseteq \text{Employees}\,(\ldots)\\ \text{Employees}\,(\text{DeptID}) &\subseteq \text{Managers}\,(\ldots)\end{aligned}$$

When attributes have the same names, we use ... on the right.

Figure 7: Example generic physical schema and dependencies.

The collection rule allows the inference of new inclusion dependencies. Assume the **EmpProjects** relation also contained the **DeptID** attribute. We could then express the following dependencies:

$$\begin{aligned} \text{EmpProjects}\,(\text{EmpID}, \text{DeptID}) &\subseteq \text{Employees}\,(\ldots) \\ \text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots) \\ \text{Employees} : \text{EmpID} &\to \text{DeptID} \end{aligned}$$

From this, we could infer the new inclusion dependency

$$\text{EmpProjects}\,(\text{EmpID}, \text{EmpName}, \text{DeptID}) \subseteq \text{Employees}\,(\ldots)\,.$$

This can be seen as collecting all attributes corresponding to a single logical entity. As we will see by example, this allows the elimination of attributes and relations via the `Fold` algorithm to reduce the size of the resulting schema while maintaining the same semantic information.

There is no finite complete axiomatization for FDs and INDs taken together [5]. Our `Expand` procedure, which uses only Mitchell's pullback and collection rules for combined inference from FDs and INDs, is sound but incomplete. However, it does terminate, since the universe of dependencies is finite and the inference process is purely additive. Although `Expand` may fail to infer some dependencies that are implied by the given set of FDs and INDs, it is nonetheless able to infer dependencies that are useful for schema design.

4.2 BCNF Decomposition

The second step, `BCNFDecompose`, is to perform a lossless join BCNF decomposition of the physical schema using the expanded set of FDs. We use a procedure similar to the one described by Garcia-Molina et al. [11].

When relations are decomposed, we project the FDs and INDs from the original relation to each of the relations resulting from decomposition. In addition, we add new inclusion dependencies which represent the correspondence of attributes between the decomposed relations. For example, when performing the decomposition $R\,(ABC) \to R'\,(AB)\,, R''\,(BC)$ we also add the INDs $R'\,(B) \subseteq R''\,(B)$ and $R''\,(B) \subseteq R'\,(B)$. In Appendix B, we prove the soundness of these dependency inferences.

In our running example, we are left with the relations and dependencies shown in Figure 8 after the `Expand` and `BCNFDecompose` steps. The **Employees** relation has been

Physical Schema

$$\begin{aligned}
\text{Employees}\,(\underline{\text{EmpID}}, \text{EmpName}, \text{DeptID}) \qquad & \text{Departments}\,(\underline{\text{DeptID}}, \text{DeptName}) \\
\text{EmpProjects}\,(\underline{\text{EmpID}}, \underline{\text{ProjID}}) \qquad & \text{EmpProjects}'\,(\underline{\text{EmpID}}, \text{EmpName}) \\
\text{Projects}\,(\underline{\text{ProjID}}, \text{ProjName}) \qquad & \text{Managers}\,(\underline{\text{DeptID}}, \text{EmpID})
\end{aligned}$$

Functional Dependencies

$$\begin{aligned}
\text{Employees} : \text{EmpID} \to \text{EmpName}, \text{DeptID} \qquad & \text{Departments} : \text{DeptID} \to \text{DeptName} \\
\text{Managers} : \text{DeptID} \to \text{EmpID} \qquad & \text{EmpProjects}' : \text{EmpID} \to \text{EmpName} \\
\text{Projects} : \text{ProjID} \to \text{ProjName}
\end{aligned}$$

Inclusion Dependencies

$$\begin{aligned}
\text{EmpProjects}\,(\text{EmpID}) &\subseteq \text{Employees}\,(\ldots) \\
\text{EmpProjects}'\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots) \\
\text{EmpProjects}'\,(\text{EmpID}) &= \text{EmpProjects}\,(\ldots) \\
\text{Projects}\,(\text{ProjID}) &= \text{EmpProjects}\,(\ldots) \\
\text{Employees}\,(\text{DeptID}) &\subseteq \text{Departments}\,(\ldots) \\
\text{Managers}\,(\text{DeptID}) &\subseteq \text{Departments}\,(\ldots)
\end{aligned}$$

Figure 8: Relations and dependencies after BCNF decomposition.

Note that $=$ is used to represent bidirectional inclusion dependencies.

decomposed to add **Departments**. Also, the **EmpProjects** relation has been decomposed to add **EmpProjects**′ and **Projects**. For illustrative purposes, we have manually given new relations sensible names. In practice, the user would need to choose relation names once the normalization process is complete.

4.3 Folding

Casanova and de Sa term the technique of removing redundant relations *folding* in the context of conceptual schema design [4]. Our algorithm, `Fold` (Figure 9), identifies any attributes or relations which are recoverable from other relations, based on the INDs. These attributes and relations are redundant and the `Fold` algorithm removes them from the

```
Function Fold(R, I) is
    Data: A set of relations R, FDs F, and INDs I
    Result: A new set R' without redundant attributes/relations
    R' ← R
    do
        // Remove redundant attributes
        foreach IND R(A) ⊆ S(B) in I do
            // Find FDs implying attributes in R are redundant
            foreach FD C → D | CD ⊆ A in F do
                // Remove attributes which are in the RHS of the FD
                R' ← R' \ {R} ⋃ {R(A \ D)}
        // Remove redundant relations
        foreach IND pair R(A) = S(B) in I do
            if R(A) = R then
                // All attributes are also in the other relation
                R' ← R' \ R
            if Pk(R(A)) = Pk(S(B)) then
                // Merge R and S
                T ← A ⋃ B
                if R' \ {R, S} ⋃ {T} is in BCNF then
                    R' ← R' \ {R, S} ⋃ {T}
    until R' is unchanged from the previous iteration;
```

Figure 9: Relation folding based on INDs

schema. More abstractly, folding removes attributes which can be recovered by joining with another relation and relations which are redundant because they are simply a projection of other relations. `Fold` also identifies opportunities for merging relations that share a common key.

It is not necessary to perform the `Fold` step to ensure that the resulting schema is in interaction-free IDNF. However, if two schemas contain equivalent information, we believe the smaller is more useful as it is a more concise representation of the application domain and does not result in any loss of information. We do not make any claims that `Fold` produces a *minimal* schema in interaction-free IDNF, but the opportunities for reduction we have identified are useful in practice. For example, consider the **EmpProjects′** relation

which contains the **EmpName** attribute. Since we have the inclusion dependency

$$\text{EmpProjects}'\,(\text{EmpID}, \text{EmpName}) \subseteq \text{Employees}\,(\ldots)$$

and the functional dependency

$$\text{Employees} : \text{EmpID} \to \text{EmpName}$$

we can infer that the **EmpName** attribute in **EmpProjects′** is redundant since it can be recovered by joining with the **Employees** relation. With the **EmpName** attribute removed, we see that we have the inclusion dependency

$$\text{EmpProjects}'\,(\text{EmpID}) = \text{EmpProjects}\,(\ldots)\,.$$

Since **EmpProjects′** is simply a projection of an attribute from **EmpProjects**, the **EmpProjects′** relation can be removed. It is important to note that the inclusion dependencies are bidirectional so that the exact set of tuples represented by the relation being removed is recoverable.

Finally, we consider an example of merging. Suppose the original schema contained another relation which stores the addresses of all employees, EmpAddress $\big(\underline{\text{EmpID}}, \text{Address}\big)$. Assuming we had an address for each employee, we can express the inclusion dependency

$$\text{Employees}\,(\text{EmpID}) = \text{EmpAddress}\,(\ldots)\,.$$

We can then merge **Employees** and **EmpAddress** by adding the **Address** attribute to the **Employees** relation.

Lemma 1. *`Fold` does not introduce any BCNF violations.*

Proof. When removing relations with `Fold`, clearly no BCNF violations are created. The attributes removed by `Fold` are never keys of the relation, so they also do not introduce BCNF violations. When attempting to merge relations via `Fold` we explicitly avoid merging if a BCNF violation would be introduced. □

4.4 Breaking IND Cycles

Mannila and Räihä [19] use a technique, which we call `BreakCycles` (Figure 10), to break circular inclusion dependencies when performing logical database design. We adopt this technique to break inclusion dependency cycles which are not proper circular.

Function `BreakCycles(`$\boldsymbol{R}$`,` $\boldsymbol{I}$`)` **is**

Data: A set of relations $\mathbf{R}$ and INDs $\mathbf{I}$

Result: A set of relations $\mathbf{R}'$ with cycles removed and new dependencies $\mathbf{I}^+$

$\mathbf{R}' \leftarrow \mathbf{R}$

foreach *Set of circular INDs* $R_1(X_1) \subseteq R_2(Y_2) \cdots \subseteq R_n(X_n) \subseteq R_1(Y_1)$ *in* $\boldsymbol{I}$ **do**

$R_1' \leftarrow X_1 Y_1$

$R_1'' \leftarrow Y_1 + \text{attr}(R_1) - X_1 Y_1$

$\mathbf{R}' \leftarrow \mathbf{R}' \setminus \{R_1\} \bigcup \{R_1', R_1''\}$

$I^+ \leftarrow I^+ \bigcup R_1'(X_1) \subseteq R_2(Y_2)$

$I^+ \leftarrow I^+ \bigcup R_1'(Y_1) \subseteq R_1''(Y_1)$

$I^+ \leftarrow I^+ \bigcup R_n(X_n) \subseteq R_1''(Y_1)$

$I^+ \leftarrow I^+ \setminus \{R_1(X_1) \subseteq R_2(Y_2), \subseteq R_n(X_n) \subseteq R_1(Y_1)\}$

Figure 10: Breaking circular inclusion dependencies

In our running example, we have an inclusion dependency cycle which is not proper circular created by the following two INDs:

$$\text{Managers}(\text{EmpID}) \subseteq \text{Employees}(\ldots)$$
$$\text{Employees}(\text{DeptID}) \subseteq \text{Managers}(\ldots).$$

Applying the `BreakCycles` algorithm removes **DeptID** from the **Employees** relation and adds a new relation WorksIn(EmpID, DeptID). We then add the following inclusion dependencies to the **WorksIn** relation:

$$\text{WorksIn}(\text{EmpID}) \subseteq \text{Employees}(\ldots)$$
$$\text{WorksIn}(\text{DeptID}) \subseteq \text{Managers}(\ldots).$$

The inclusion dependency Employees(DeptID) $\subseteq$ Managers(...) is also removed, breaking the cycle.

Lemma 2. *`BreakCycles` does not introduce any BCNF violations.*

Proof. `BreakCycles` decomposes a relation into two relations with the only functional dependency defined establishing a primary key. The only new dependencies added are inclusion dependencies between corresponding attributes in the decomposed relations. This does not permit the inference of any new functional dependencies. Therefore, the final schema is still in BCNF with respect to $\mathbf{F}'$. □

4.5 IDNF

The goal of our normalization algorithm is to produce a schema that is in interaction-free IDNF with respect to the given dependencies. The following conditions are sufficient to ensure that a set of relations **R** is in interaction-free IDNF with respect to a set of FDs **F** and INDs **I**:

1. **R** is in BCNF [8] with respect to **F**.
2. All the INDs in **I** are key-based or proper circular.
3. **F** and **I** do not interact.

A set of INDs is proper circular if for each circular inclusion dependency over a unique set of relations $R_1(X_1) \subseteq R2(Y_2), R_2(X_2) \subseteq R_3(Y_3), \ldots, R_m(X_m) \subseteq R_1(Y_1)$, we have $X_i = Y_i$ for all i.

Lemma 3. *The schema produced by the normalization algorithm of Figure 6 is in interaction-free IDNF with respect to the given sets of FDs and INDs.*

Proof. Rule 1 is satisfied because `BCNFDecompose` produces a schema that is BCNF with respect to $\mathbf{F}'$, and therefore with respect to **F**. Furthermore, the subsequent `Fold` and `BreakCycles` algorithms do not introduce any BCNF violations.

Rule 2 states that all remaining INDs must be key-based. The given set of INDs (**I**) is superkey-based by assumption. We can show that all of the additional INDs created by the algorithm are also key-based. Furthermore, none of the schema transformations can result in non-key-based INDs. A complete proof of this is given in Appendix D.

To show that the final schema satisfies rule 3 (non-interaction of FDs and INDs), we make use of a sufficient condition for non-interaction given by Levene [18]. A set of FDs **F** and INDs **I** over a set of relations do not interact if the relations are in BCNF with respect to **F**, **I** is proper circular, and $\mathbf{F} \bigcup \mathbf{I}$ is *reduced*. As stated above, the final schema is in BCNF. All inclusion dependencies are proper circular since we explicitly break any cycles which are not via the `BreakCycles` algorithm. It remains to show that the set of functional and inclusion dependencies are reduced.

A set of functional inclusion dependencies **F** and **I** is reduced if for every inclusion dependency $R(X) \subseteq S(Y)$, there are only trivial functional dependencies involving attributes in the set Y. We have already shown that the final set of inclusion dependencies is key-based, implying that Y is a key of S. Since the set Y is a key, **F** can only contain trivial functional dependencies involving Y. Therefore, $\mathbf{F} \bigcup \mathbf{I}$ is reduced and the schema is in interaction-free IDNF. □

5 Dependency Mining

As we noted in Section 3, the dependencies required by our algorithm can be directly specified by a knowledgeable user, or mined from an instance of the underlying database. In this section, we consider the latter option in more depth.

5.1 Mining for NoSQL Normalization

Dependency mining tools operate by examining a database instance to discover all dependencies that hold on it. In order to mine dependencies, such tools collect statistics on the distribution of values in each column in an attempt to discover relationships. We make use of Apache Calcite [1] to provide an interface between NoSQL databases and dependency mining tools, so that the tools can obtain the metadata and statistics they require. Calcite is a data management framework which presents a SQL query interface on top of a variety of database backends. We have used TANE [15] for mining functional dependencies and BINDER [23] for mining inclusion dependencies. These algorithms both produce all valid dependencies which hold on the given instance.

The problem with using mined dependencies is that many of them will be spurious. For example, suppose in the employees relation of a company database we have the functional dependency $\text{Department}, \text{Salary} \rightarrow \text{FirstName}$. This expresses that all employees in a department with the same salary have the same first name. While this might hold for some particular instance of the schema, it is unlikely to represent semantically meaningful information. Another valid dependency on the same relation may be $\text{DeptID} \rightarrow \text{Department}$. We would prefer to perform BCNF decomposition using the second dependency since it would result a table with a primary key of DeptID, which is likely to be more useful than one with key $(\text{Department}, \text{Salary})$.

We use two techniques to reduce the impact of spurious dependencies: 1) ranking of functional dependencies for the selection of primary keys for a relation and 2) ranking of functional dependencies for the selection of a BCNF-violating dependency for decomposition. Any time we generate a new relation, we use heuristics to select a primary key. Functional dependencies representing other possible candidate keys are then ignored when performing BCNF decomposition. Instead, we use heuristics to rank the remaining dependencies to select the next violating functional dependency to use for decomposition. Note that we still consider all valid functional dependencies which violate BCNF. However, by selecting a good order for decomposition, some spurious functional dependencies no longer

apply when their attributes are split into separate relations. Avoiding decomposition based on these dependencies results in a more logical schema as output.

We make use of three heuristics to identify functional dependencies which are likely to represent keys:

1. **Length**: Keys with fewer attributes
2. **Value**: Keys with shorter values
3. **Position**: Keys occurring further left in the relation definition without non-key attributes between key attributes.

We use these heuristics for both primary key and violating dependency selection. Since we target NoSQL databases, we do not blindly apply the value length heuristic to all columns. This is because data types exist which are explicitly intended to represent unique identifiers. For example, Cassandra allows columns of type UUID and MongoDB documents can have values of type ObjectId. These are both long pseudorandom values intended to allow concurrent creation without collision. Thus, although the values are long, we know that these values are likely to represent key attributes. We assign such columns the highest score according to the Value heuristic.

Papenbrock and Naumann [24] used similar heuristics in an algorithm for BCNF normalization of a schema using mined functional dependencies. (They did not consider identifier types, since they did not target NoSQL databases.) They also propose an additional heuristic which measures duplication across sets of column values in a dependency. We did not use this heuristic since it increases complexity by requiring joint statistics across multiple columns, and our algorithm produces positive results without this heuristic.

6 Applications of the Logical Model

The logical schema produced by the renormalization process is useful as a form of documentation of the information that is embodied, in denormalized form, in a NoSQL database. However, the logical schema has other applications as well. Our original motivation for this work was to be able to provide a conceptual model of an existing NoSQL database as input to a NoSQL schema design tool, such as NoSE. Given a conceptual model of the database, as well as a description of the application workload, NoSE generates a physical schema

Logical schema query

```
SELECT EmpName, ProjID, ProjName FROM Projects
NATURAL JOIN Employees WHERE EmpName = ?
```

Physical schema query

```
SELECT EmpName, ProjID, ProjName FROM EmpProjects WHERE EmpName = ?
```

Figure 11: Query rewriting against the logical schema

optimized to support that workload. By combining renormalization with a schema design tool, we can optimize the physical schema design of an existing NoSQL-based application.

It may also be useful to express application queries and updates directly against the logical model. This can provide a means of executing new, ad-hoc queries over an existing NoSQL database without the need to understand how the data is denormalized. In the remainder of this section, we discuss how we can execute queries expressed over the logical model using information gathered during the execution of our normalization algorithm.

6.1 Ad-Hoc Query Execution

One of the main advantages of using dependency information to construct the logical schema is that we can use the same information to assist with executing queries written against the logical schema. Because NoSQL databases often lack the ability to perform any complex processing of queries, developers express queries directly in terms of structures from the physical schema. This tightly couples the application to a particular schema and makes changes in the schema difficult. With an appropriate logical schema for the application, we can rewrite queries written against this logical schema to target specific physical structures as in Figure 11. In this case, we can identify that the **EmpProjects** relation materializes the join in the logical schema query and is therefore able to provide an answer. Our aim is for this rewriting to happen transparently and to enable the possibility of changing the rewriting as the physical schema changes.

As we show in the following section, we can produce queries on the logical schema which correspond to data stored in the original structures in the physical schema. We can think of these queries as defining materialized views over the logical schema which correspond to the physical schema. The application developer can use these queries directly in cases where the application directly used data from these structures without additional manipulation. This simplifies rewriting existing application queries if a developer wishes to move to using the logical model. For more complex queries, we can use existing techniques to rewrite the

queries to make use of the materialized views [13].

These queries can be translated on-the-fly to enable ad-hoc query execution. For example, Apache Calcite [1] is a dynamic data management framework which connects to different backends, including those for NoSQL datastores. We are currently exploring rules for view-based query rewriting in Calcite to enable the necessary transformations. We leave a full implementation of this approach as future work.

6.2 View Definition Recovery

In order to allow logical queries to execute against the existing physical schema, we must have a way of understanding how the existing physical structures map to the logical schema. Fortunately, we can use information saved from the normalization process to produce this mapping. We simply think of each physical structure in the original schema as a materialized view. We can recover a query which serves as the materialized view definition by tracking a small amount of additional information during the normalization process.

For an example of view definition recovery, consider the **EmpProjects** relation from Figure 7. Before performing normalization, our set of relations is equivalent to the input so our view definition for **EmpProjects** is `SELECT EmpID, EmpName, ProjID, ProjName FROM EmpProjects`. Considering only the **EmpProjects** relation, we have the following functional dependencies:

$$\text{EmpID} \rightarrow \text{EmpName and}$$
$$\text{ProjID} \rightarrow \text{ProjName.}$$

When we perform BCNF decomposition, the normalization algorithm splits **EmpProjects** into three relations. We call the relation with employee data **EmpProjects′**, the relation with project data **Projects**, and keep the remaining relation expressing the association with the name **EmpProjects**. We can now write the view definition to include a join based on the decomposition. Our view definition then appears as below:

```
SELECT EmpID, EmpName, ProjID, ProjName FROM EmpProjects JOIN EmpProjects'
  ON EmpProjects.EmpID = EmpProjects'.EmpID
  JOIN Projects ON EmpProjects.ProjID = Projects.ProjID.
```

A similar process of creating joins applies when running the `BreakCycles` algorithm. The other transformation which affects the view definitions is `Fold`. When removing

relations, the transformation is a simple rename of the relation in the view definition. For example, after the `Fold` step of our algorithm is performed on the **EmpProjects′** relation, we see that it can be removed as was discussed in Section 4.3. This is because the data in **EmpProjects′** can be recovered from the **Employees** relation. We can simply replace all instances of **EmpProjects′** in the definition above with **Employees**. We do not show an example, but a similar renaming applies when `Fold` removes an attribute with the addition that a join is also created involving the relation which contains the removed attribute.

For a relation R, we can recover the list of logical structures it references by recursively visiting the list of relations decomposed to produce R until we reach physical structures from the original schema. Since all of our inclusion dependencies are superkey-based, all of the view definitions will consist of foreign key joins. More specifically, a materialized view definition for the relation R will be of the form `SELECT` $attr\,(R)$ `FROM` R_1 `JOIN` R_2 `ON` $R_1.A = R_2.B \cdots$ `JOIN` $R_{n-1}.X = R_n.Y$ where $attr\,(R)$ is a list of the attributes in R and R_1 through R_n are the relations the query must join.

Using these materialized view definitions, we can answer queries written against the logical schema using view-based query rewriting as discussed in the previous section. The goal of view-based query rewriting is to answer a query using a set of materialized views, which is exactly what we are trying to accomplish. We note that the view definitions we described above are all conjunctive queries. It has recently been shown that conjunctive query determinacy is undecidable in general [12]. However, there are useful subclasses of conjunctive queries for which determinacy is decidable [25].

7 Case Studies

This section presents case studies of several denormalized database schemas to show how ESON is able to recover a useful schema. We discuss both cases where dependencies were specified manually and where the dependencies were mined using an instance of the denormalized schema.

7.1 RUBiS

RUBiS [6] is a benchmark based on a Web application for online auctions. The authors also developed a schema design tool called NoSE [21], which performs automated schema design for NoSQL systems. We used NoSE to generate two Cassandra schemas for RUBiS,

each optimized for a different workload (a full description is given in Appendix E). In each case, NoSE starts with a conceptual model of the RUBiS database, The conceptual model includes six types of entities (e.g., users, and items) with a variety of relationships between them. The first physical design consists of 9 Cassandra column families, while the second, larger design has 14 column families.

As our first case study, we used NoSE's denormalized Cassandra schemas as input to our normalization algorithm so that we can compare the normalized schemas that it produces with the original conceptual schema that NoSE started with. For each physical schema, we tested our algorithm with two different sets of dependencies: one set manually generated from the physical schema, and a second set mined from an instance of that schema using the mining technique discussed in Section 5. This resulted in a total of four tests.

For both schemas, renormalization using manually identified dependencies resulted in a conceptual model that was identical (aside from names of relations and attributes) to the original conceptual schema used by NoSE, as desired.

For the two tests with mined dependencies, the renormalization program produced the original conceptual schema, as desired, in the case of the smaller (9 column family) Cassandra schema, but not in the case of the larger (14 column family) Cassandra schema. For the smaller schema, the mining process identified 61 functional dependencies and 314 inclusion dependencies. The dependency ranking heuristics were critical to this success. Without them, spurious dependencies lead to undesirable entities in the output schema. For example, one contains only the fields **BidValue** and **BidQuantity**, which is not a semantically meaningful entity. For the larger schema, mining found 86 functional dependencies and 600 inclusion dependencies, many of them spurious. In this case, the ranking heuristics were not sufficient to eliminate undesirable decompositions during renormalization. No set of ranking heuristics will be successful in all cases, but it is clear that this is an important area for improvement in future work.

The large schema test with manually chosen dependencies provided a good example of relation merging using `Fold` step of our algorithm. In the conceptual schema, there is a **Comments** entity set which has relationships to the user sending and receiving the comment. The denormalized schema has two separate relations which store the comments according to the sending and receiving users. After performing BCNF decomposition, we end up with relations similar to the following (simplified for presentation):

$$\text{CommentsSent}\,(\text{id}, \text{sending\_user}, \text{text})$$
$$\text{CommentsReceived}\,(\text{id}, \text{receiving\_user})\,.$$

We also have inclusion dependencies which specify that the comment_id attribute in both relations is equivalent, i.e. $\text{CommentsSent}\,(\text{id}) = \text{CommentsReceived}\,(\ldots)$. Since the key

Publishers(_id, name, founded, book)
Books(_id, title, author)
Patrons(_id, name, address.city, address.state,
loans._id, loans.title,
loans.author._id, loans.author.name)
Authors(_id, name)

Figure 12: Physical relations from MongoDB schema

of these relations is equivalent, the `Fold` algorithm will merge these two relations producing Comments(id, receiving_user, sending_user, text).

These examples show that functional and inclusion dependencies are able to drive meaningful denormalization. Runtime for the normalization step of our algorithm was less than one second on a modest desktop workstation in all cases.

7.2 MongoDB

Stolfo [26] presents a case study of schema design in MongoDB to explore design alternatives. We extract a design from the examples to show how our normalization process can produce a suitable logical model. The system being designed is for library management and deals with patrons, books, authors, and publishers. While the case study shows different possible schemas, we have selected one for demonstration purposes and we present example documents for this schema below. This model contains a significant amount of denormalized data inside the collection of patron documents.

As described in Section 2, we first defined a physical relational schema capturing the information in the MongoDB database documents. This is shown in Figure 12. The MongoDB patron documents included an array of book loans for each patron. To produce the physical schema, we flattened the loan attributes into the **Patron** relation, and added the key of each array element as part of the superkey the **Patron** relation. We also manually identified a (non-exhaustive) set of functional and inclusion dependencies over these relations, as shown in Figure 13.

The denormalization in this schema consists of the duplication of patron, book, and author information in the patron documents. For this application, the algorithm was able

$$
\begin{aligned}
\text{Publishers} &: \text{\_id} \rightarrow \text{name, founded, book} \\
\text{Books} &: \text{\_id} \rightarrow \text{title, author} \\
\text{Patrons} &: \text{\_id} \rightarrow \text{name, address.city,} \\
&\qquad \text{address.state, loans.\_id} \\
\text{Authors} &: \text{\_id} \rightarrow \text{name}
\end{aligned}
$$

$$
\begin{aligned}
\text{Publishers(book)} &\subseteq \text{Books (\_id)} \\
\text{Books(author)} &\subseteq \text{Authors (\_id)} \\
\text{Patrons(loans.\{\_id, title,} & \\
\text{author.\_id\})} &\subseteq \text{Books (\_id, title, author)} \\
\text{Patrons(loans.author.} & \\
\text{\{\_id, name\})} &\subseteq \text{Authors (\_id, name)}
\end{aligned}
$$

Figure 13: Dependencies on MongoDB physical relations

to identify the denormalization and produce a logical model without duplication. The logical schema produced by our normalization algorithm removes this denormalization. We note that the dependencies in Figure 13 do not contain any functional dependencies involving loans, which were nested in patron documents in MongoDB database. However, the `Expand` step of our algorithm is able to infer such functional dependencies based on the inclusion dependencies between **Patrons** and **Authors**/**Books** and the FDs on those relations. The `BCNFDecompose` step separates the redundant title and author information from the **Patrons** relation using these FDs. Finally, the `Subsume` step removes this data since it is duplicated in the **Authors** and **Books** relations. This removes all redundancy which was present in the original schema in Figure 12. Relationships between publishers, authors and their books were also recovered. The final schema represented as an ER diagram is shown in Figure 14.

7.3 Twissandra

Twissandra [10] is a simple clone of the Twitter microblogging platform using Cassandra as a database backend. The application stores data on only two different entities: users and tweets. Each tweet has an associated user who created the tweet and each user can

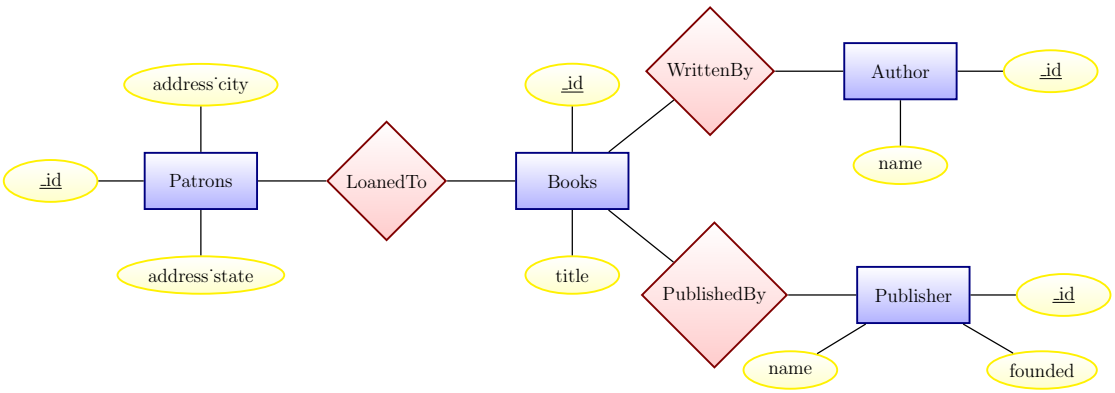

Figure 14: MongoDB example schema entities

$$
\begin{aligned}
\text{users}\,(\underline{\text{uname}}, \text{password})\\
\text{following}\,(\underline{\text{uname}}, \underline{\text{followed}})\\
\text{followers}\,(\underline{\text{uname}}, \underline{\text{following}})\\
\text{tweets}\,(\underline{\text{tweet\_id}}, \text{uname}, \text{body})\\
\text{userline}\,(\underline{\text{tweet\_id}}, \underline{\text{uname}}, \text{body})\\
\text{timeline}\,(\underline{\text{uname}}, \underline{\text{tweet\_id}}, \text{posted\_by}, \text{body})
\end{aligned}
$$

Figure 15: Twissandra physical relations

"follow" any number of other users.

The Twissandra schema consists of six column families. There is one for both users and tweets keyed by their respective IDs. Two additional column families store the users a particular user is following and separately, their followers. Denormalizing this relationship allows efficient retrieval of users in both directions. Finally, there is a column family storing all data on tweets by user and a column family which stores tweets for all users a user is following. Both of these final two column families contain denormalized data on tweets. The relations corresponding to these column families are given in Figure 15.

Figure 16 shows all the functional and inclusion dependencies which we can express over the Twissandra schema. Note that we use $=$ to denote a bidirectional inclusion dependency and we omit attribute names from the right-hand side of inclusion dependencies when the attribute names are the same as on the left-hand side.

$$
\begin{aligned}
\text{tweets} : \text{tweet\_id} &\to \text{uname}, \text{body} \\
\text{userline} : \text{tweet\_id} &\to \text{uname}, \text{body} \\
\text{timeline} : \text{tweet\_id} &\to \text{posted\_by}, \text{body} \\
\text{users} : \text{uname} &\to \text{password}
\end{aligned}
$$

$$
\begin{aligned}
\text{followers}\,(\text{uname}, \text{following}) &= \text{following}\,(\text{followed}, \text{uname}) \\
\text{userline}\,(\text{uname}) &\subseteq \text{users}\,(\ldots) \\
\text{timeline}\,(\text{uname}) &\subseteq \text{users}\,(\ldots) \\
\text{timeline}\,(\text{posted\_by}) &\subseteq \text{users}\,(\text{uname}) \\
\text{timeline}\,(\text{uname}, \text{posted\_by}) &\subseteq \text{following}\,(\ldots, \text{followed}) \\
\text{timeline}\,(\text{posted\_by}, \text{uname}) &\subseteq \text{followers}\,(\text{following}, \ldots) \\
\text{userline}\,(\text{tweet\_id}, \text{uname}, \text{body}) &= \text{tweets}\,(\ldots)
\end{aligned}
$$

Figure 16: Dependencies on Twissandra physical relations

Figure 17 shows a complete ER diagram for Twissandra, which represents the desired result. However, the conceptual model produced by our normalization process includes one additional relation aside from users and tweets. The conceptual schema produced by our algorithm is still fully normalized in interaction-free IDNF. The additional relation occurs because the normalization process is unable to remove the timeline column family, despite the fact that the information contained in this column family is still redundant. We can reconstruct the timeline relation with the query

```
SELECT f.uname, t.tweet_id, t.uname AS posted_by, t.body
FROM following f JOIN tweets t ON t.followed = t.uname.
```

This redundancy remains because the dependency defining the timeline table cannot be expressed using functional or inclusion dependencies. Expressing this denormalization requires a dependency language which can reference more than two relations. In this case, the dependency is between timeline, followers, and tweets. Extending our dependency language and normalization process to include additional dependencies such as multivalued dependencies [9] would enable us to resolve such issues.

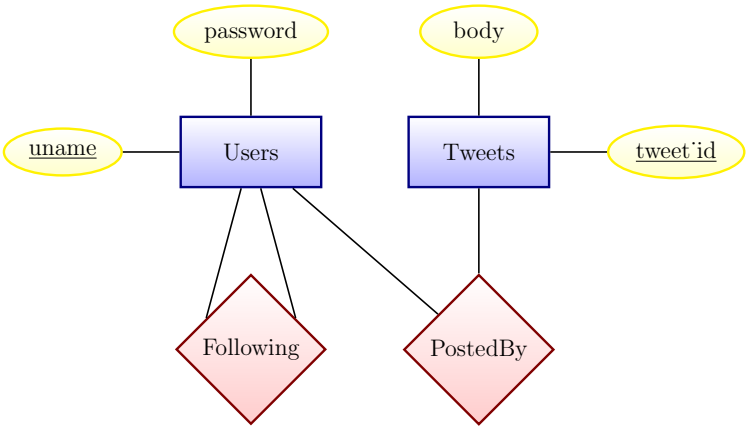

Figure 17: Twissandra schema entities

8 Conclusion

We have developed a methodology for transforming a denormalized physical schema in a NoSQL datastore into a normalized logical schema. Our method makes use of functional and inclusion dependencies to remove redundancies commonly found in NoSQL database designs. We further showed how we can make use of dependencies which were mined from a database instance to reduce the input required from users. Our method has a variety of applications, such as enabling query execution against the logical schema and guiding schema evolution and database redesign as application requirements change.

There are additional opportunities for further automation of NoSQL schema management tasks. One limitation of the logical schema produced by our algorithm is that the relations are not necessarily given meaningful names. Some heuristics such as looking for common prefixes in attribute names may be useful. Currently we also require developers to modify applications manually after we produce the logical model. However, previous work such as Query By Synthesis [7] has shown that it is possible to extract higher-level query patterns from imperative application code. A similar approach could be applied to extract queries from applications which could then be rewritten to use the logical model. We also discussed the possibility of a query execution engine which could transparently retarget these queries to operate on different physical models. We expect a combination of these techniques to improve schema management for NoSQL databases.

References

[1] Apache Calcite, 2018. Retrieved Jun. 14, 2018 from https://calcite.apache.org.

[2] William Ward Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[3] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 2016.

[4] Marco A. Casanova and Jose E. Amaral de Sa. Mapping uninterpreted schemes into entity-relationship diagrams: Two applications to conceptual schema design. *IBM Journal of Research and Development*, 28(1):82–94, Jan 1984.

[5] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

[6] Emmanuel Cecchet et al. Performance and scalability of EJB applications. *ACM SIGPLAN Notices*, 37(11):246–261, 2002.

[7] Alvin Cheung et al. Optimizing database-backed applications with query synthesis. In *PLDI '13*, pages 3–14, Seattle, WA, USA, 2013.

[8] Edgar F. Codd. Recent investigations into relational data base systems. Technical Report RJ1385, IBM, Apr 1974.

[9] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM TODS*, 2(3):262–278, Sep 1977.

[10] Eric Florenzano, Tyler Hobbs, Eric Evans, et al. Twissandra. Retrieved Jun. 14, 2018 from https://github.com/twissandra/twissandra.

[11] Hector Garcia-Molina et al. *Database systems: the complete book*. Pearson Prentice Hall, Upper Saddle River, N.J., 2 edition, 2009.

[12] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *LICS '15*, pages 281–292, Kyoto, Japan, 2015. IEEE Computer Society.

[13] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, May 2001.

[14] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Data schema does matter, even in NoSQL systems! In *RCIS '16*, Grenoble, France, June 2016.

[15] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[16] Javier Luis Cánovas Izquierdo and Jordi Cabot. *Discovering Implicit Schemas in JSON Data*, pages 68–83. Springer, Berlin, Heidelberg, Jul 2013.

[17] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In *BTW 2015*, Hamburg, Germany, 2015.

[18] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE TKDE*, 12(2):281–291, Mar 2000.

[19] Heikki Mannila and Kari-Jouko Räihä. Inclusion dependencies in database design. In *ICDE '86*, pages 713–718, Los Angeles, CA, USA, Feb 1986.

[20] Christopher J. Matheus et al. Systems for knowledge discovery in databases. *IEEE Transactions on knowledge and data engineering*, 5(6):903–913, 1993.

[21] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. In *ICDE '16*, pages 181–192, Helsinki, Finland, May 2016.

[22] John C. Mitchell. *Inference Rules for Functional and Inclusion Dependencies*, pages 58–69. PODS '83. ACM, 1983.

[23] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. In *VLDB '15*, volume 8, pages 774–785, Hawaii, USA, February 2015.

[24] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *EDBT '17*, pages 342–353, 2017.

[25] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *ICDT '11*, pages 220–231, Uppsala, Sweden, 2011.

[26] Emily Stolfo. MongoDB schema design, 2013. Retrieved Jun. 14, 2018 from https://www.slideshare.net/mongodb/mongodb-schema-design-20356789.

[27] Lanjun Wang et al. Schema management for document stores. In *VLDB '15*, volume 8, pages 922–933, Hawaii, USA, May 2015.

Appendix A ESON Proofs

Each of the proofs in the section below consists of a claim about the normalization algorithm in Figure 6. The steps are the proof are given according to each step in the algorithm.

Appendix B All inference of dependencies is sound

`Expand`

When expanding $\mathbf{F}$ and $\mathbf{I}$ we use axioms presented by Mitchell [22] which are shown to be sound.

`BCNFDecompose`

When performing BCNF decomposition, we add two new inclusion dependencies which state the equivalence of attributes in the decomposed relations. For example, if we decompose $R(A, B, C)$ into $S(A, B)$ and $T(B, C)$ then we would add the inclusion dependencies $S(B) \subseteq T(B)$ and $T(B) \subseteq S(B)$. These inclusion dependencies hold by construction.

We also project any FDs and INDs onto the new tables. Any projected FDs are already contained in $\mathbf{F}'$ as a result of the axiom of reflexivity. Similarly, the projected INDs are already contained in $\mathbf{I}'$ aside from a simple renaming of relations to the new names after decomposition. For example, if we decompose $R(A, B, C)$ into $R'(A, B)$ and $R''(B, C)$ then we consider inclusion dependencies involving R using the attributes A and B and change them to reference R'. These hold since R' contains exactly the same (A, B) tuples as R. A similar argument holds for inclusion dependencies on R containing the attributes B and C.

`Fold`

When removing attributes, all relevant dependencies are already contained in $\mathbf{F}'$ and $\mathbf{I}^{+'}$ since they are projections of other dependencies. When removing relations, there is no need for any new dependencies and we simply remove any dependencies referencing the removed relation from $\mathbf{I}^{+'}$. Finally, when merging relations we simply rename existing inclusion dependencies to reference the merged relation. If we merge $R(A, B)$ and $S(A, C)$

into $RS\,(A, B, C)$ then we consider separately inclusion dependencies on R involving A and B and inclusion dependencies on S involving A and C. RS contains exactly the same (A, B) tuples as R so any inclusion dependencies involving R will also hold on RS. A similar argument applies for inclusion dependencies involving S and the attributes A and C.

`BreakCycles`

Our technique for breaking cycles is taken from Mannila and Räihä [19]. Suppose we have a cycle $R_1\,(X_1) \subseteq R_2\,(Y_2) \cdots \subseteq R_n\,(X_n) \subseteq R_1\,(Y_1)$. R_1 is decomposed into $R'_1\,(X_1 \bigcup Y_1)$ and $R''_1\,(Y_1 \setminus (\mathrm{attr}\,(R_1) \setminus X_1 Y_1))$. Three inclusion dependencies are added. $R'_1\,(X_1) \subseteq R_2\,(Y_2)$ which is derived be renaming R to R_1 which is sound since these relations contain the same tuples when X_1 is projected. The same argument holds for the inclusion dependency $R_n\,(X_n) \subseteq R''_1\,(Y_1)$. Finally, the transformation also adds the inclusion dependency $R'_1\,(Y_1) \subseteq R''_1\,(Y_1)$. Since R'_1 and R''_1 both contain values for Y_1 decomposed from R, this dependency also holds. The original authors also show that the transformation is information-preserving.

Appendix C All transformations are lossless-join

`BCNFDecompose`

We use a known algorithm for lossless-join BCNF decomposition.

`Fold`

When `Fold` removes an attribute B from a relation $R\,(A, B)$ it is because there exists a relation $S\,(C, D)$ with an inclusion dependency $R\,(A, B) \subseteq S\,(C, D)$. In this case, we would add the relation $R'\,(A)$ to the schema and remove R. Note that R can be reconstructed by joining R with S on $A = C$ and projecting A and D (renamed to B), that is $\rho_{B/D}\,(\Pi_{A,D}\,(R \bowtie S))$.

If a relation R is removed by `Fold`, it is because there is a bidirectional dependency with a relation S indicating that there is a one-to-one mapping between records in R and records in S. Therefore, we can remove R since it can be recovered by a simple projection of S.

Finally, `Fold` can merge two relations with a common key. If we merge $R\,(A, B)$ and $S\,(A, C)$ to form $T\,(A, B, C)$ we can recover R and S by simply projecting the appropriate fields from T.

`BreakCycles`

When breaking an inclusion dependency cycle, each of the new relations contains a key of the decomposed relation. Therefore, we can perform a natural join on this key to produce the original relation.

Appendix D Inclusion dependencies in the final schema are key-based

`Expand`

Two inference rules can generate new INDs. The first is implication via transitivity. If we have functional dependencies $R(X) \subseteq S(Y)$ and $S(Y) \subseteq T(Z)$ then we can infer $R(X) \subseteq T(Z)$. Since we assume existing INDs are superkey-based, Z must be a superkey of T and the new IND introduces no violations. We can also infer new INDs by exploiting the set of functional and inclusion dependencies together. Suppose we have the INDs $R(X_1) \subseteq S(Y_1)$ and $R(X_2) \subseteq S(Y_2)$ as well as the functional dependency $R : X_1 \to X_2$. Then we can infer the IND $R(X_1X_2) \subseteq S(Y_1Y_2)$. Since Y_1 and Y_2 are both superkeys of Y, this new IND is also superkey-based. The second is the collection rule. Since the right-hand side of INDs created by the collection rule is a superset of an existing IND, the new IND is also superkey-based.

`BCNFDecompose`

Assume we have a relation $R(XYZ)$ where X is the key of R and also the functional dependency $R : Y \to Z$. We would then decompose R into $R'(XY)$ and $R''(YZ)$. Suppose we had another relation S with an IND $S(A) \subseteq R(X)$. After decomposing R we would create the IND $S(A) \subseteq R'(X)$ which by construction is also superkey-based since X is the key of R'. The situation is slightly more complicated if we had an IND $S(UV) \subseteq R(XZ)$. After decomposition X and Z are in separate relations. We then have the INDs $S(U) \subseteq R'(X)$ and $S(V) \subseteq R''(Z)$. However, Z is not a superkey of R''. When this situation occurs, we drop the inclusion dependency $S(V) \subseteq R''(Z)$. Note that these inclusion dependencies are not necessary to satisfy IDNF but they are useful to identify possible foreign keys in the final schema.

`Fold`

`Fold` converts INDs which are superkey-based into key-based INDs. Assume we have an IND $R(AB) \subseteq S(CD)$ where C is the key of S. Then this IND is not key-based. However,

since C is a key of S, we must have the functional dependency $C \rightarrow D$. Therefore, the `Fold` algorithm will identify the attribute B as redundant and remove it from R. This changes the inclusion dependency to $R(A) \subseteq S(C)$ which is now key-based.

`BreakCycles`

Suppose we have a cycle of the form $R_1(X_1) \subseteq R_2(Y_2) \cdots \subseteq R_n(X_n) \subseteq R_1(Y_1)$. As discussed in 4.1, breaking this circularity will result in new relations R'_1 and R''_1. There are also new INDs $R'_1(X_1) \subseteq R_2(Y_2)$, $R'_1(Y_1) \subseteq R''_1(Y_1)$, and $R_n(X_n) \subseteq R''_1(Y_1)$. Given that the given dependencies (resulting from the `Fold` step) are key-based, we note that Y_2 is a key of R_2. In addition, since we construct R'_1 and R''_1 such that Y_1 is a key, all new INDs are also key-based.

Appendix E ESON RUBiS Example Input

Physical relations used as input for both examples are given here. Relation names are shown in bold an attributes which are keys are underlined.

E.1 Schema One

i154863668(categories_id, items_end_date, items_id, regions_id, users_id, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i1888493477(items_id, items_buy_now, items_description, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids, items_quantity, items_reserve_price, items_start_date)
i193173044(bids_date, bids_id, items_id, users_id, bids_bid, bids_qty, users_nickname)
i2906147889(users_id, users_balance, users_creation_date, users_email, users_firstname, users_lastname, users_nickname, users_password, users_rating)
i3157175159(comments_id, users_id, comments_comment, comments_date, comments_rating)
i3220017915(bids_id, items_id, bids_bid, bids_date, bids_qty)
i3722443462(categories_id, categories_name)
i546546186(categories_id, items_end_date, items_id, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i590232953(regions_id, regions_name)

E.2 Schema Two

i1177375268(buynow_id, buynow_date, items_id)
i1557291277(users_id, comments_id, comments_comment, comments_date, comments_rating)
i1879743023(comments_id, users_id, users_nickname)
i2049737091(items_id, bids_id, users_id, items_end_date)
i2087519603(items_id, categories_id)
i210798434(items_id, bids_date, bids_id, users_id, bids_bid, bids_qty, users_nickname)
i2269844981(items_id, users_id, items_end_date)
i2366332486(users_id, buynow_date, buynow_id, buynow_qty)
i2594090645(items_id, items_buy_now, items_description, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids, items_quantity, items_reserve_price, items_start_date)
i262196338(items_id, bids_bid, bids_id, bids_date, bids_qty)
i3050485475(categories_id, categories_name)
i3116489164(users_id, users_balance, users_creation_date, users_email, users_firstname, users_lastname, users_nickname, users_password, users_rating)
i409321726(users_id, items_end_date, bids_id, items_id)
i920605840(categories_id, items_id, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i941409494(users_id, items_end_date, items_id)