Renormalization of NoSQL Database Schemas

Michael J. Mior(✉) and Kenneth Salem

Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
{mmior,kmsalem}@uwaterloo.ca

Abstract. NoSQL applications often use denormalized databases in order to meet performance goals, but this introduces complications as the database itself has no understanding of application-level denormalization. In this paper, we describe a procedure for reconstructing a normalized conceptual model from a denormalized NoSQL database. The procedure's input includes functional and inclusion dependencies, which may be mined from the NoSQL database. Exposing a conceptual model provides application developers with information that can be used to guide application and database evolution.

Keywords: Renormalization · NoSQL · Database design
Conceptual modeling

1 Introduction

NoSQL databases, such as Apache Cassandra, Apache HBase, and MongoDB, have grown in popularity, despite their limitations. This is due in part to their performance and scalability, and in part because they adopt a flexible approach to database schemas. Because these systems do not provide high-level query languages, applications must denormalize and duplicate data across physical structures in the database to answer complex queries. Unfortunately, the NoSQL database itself typically has no understanding of this denormalization. Thus, it is necessary for applications to operate directly on physical structures, coupling applications to a particular physical design.

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 [6]. For example, consider an application using HBase to track requests made to an on-line service. The same requests may be stored in multiple tables since the structures available determine 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. Since the NoSQL system itself is unaware of these application decisions, it can provide little to no help in understanding what is being represented in the database.

J. C. Trujillo et al. (Eds.): ER 2018, LNCS 11157, pp. 479–487, 2018.
https://doi.org/10.1007/978-3-030-00847-5_34

Together, the lack of physical data independence and the need for workload-tuned, denormalized database designs creates challenges for managing and understanding physical schemas, especially as applications evolve. In our online 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. Unless the application developer maintains external documentation, the only knowledge of this denormalization is embedded within the source code. We aim to surface this knowledge by generating a useful conceptual model of the data.

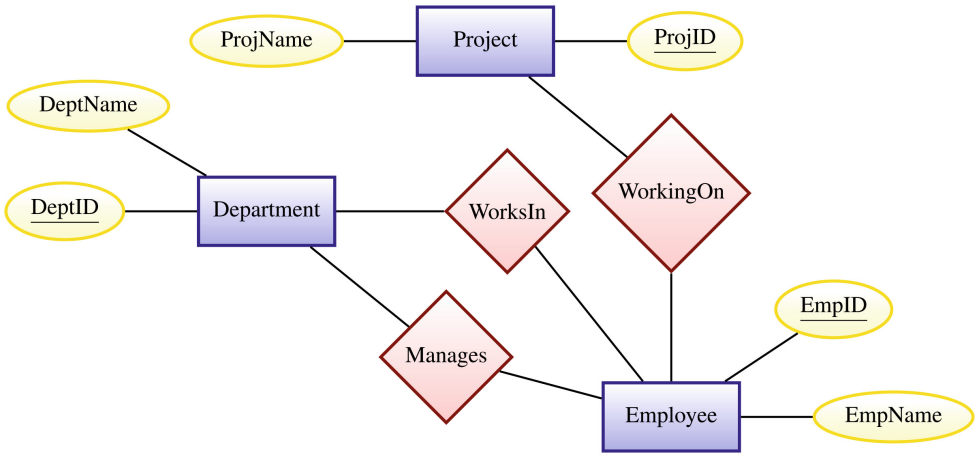

Fig. 1. Schema example after renormalization

We refer to this surfacing task as *schema renormalization*. This work addresses the schema renormalization problem through the following technical contributions:

- We present a semi-automatic technique for extracting a normalized conceptual model from an existing denormalized NoSQL database. It produces a normalized conceptual model for the database, such as the one shown in Fig. 1.
- We develop an normalization algorithm in Sect. 5 which forms the core of our approach. This algorithm uses data dependencies to extract a conceptual model from the NoSQL system's physical structures. Our algorithm ensures that the resulting model is free from redundancy implied by these dependencies. To the best of our knowledge, this is the first normalization algorithm to produce a schema in interaction-free inclusion dependency normal form [9].
- Finally, Sect. 6, presents a case study which shows the full schema renormalization process in action for a NoSQL application. We use this case study to highlight both the advantages and the limitations of our approach to 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.

2 Renormalization Overview

We renormalize NoSQL databases using a three step process. The first step is to produce a *generic physical schema* describing the physical structures that are present in the NoSQL database. We describe the generic physical model in more detail in Sect. 3, and illustrate how it can be produced for different types of NoSQL systems. The second step is to identify dependencies among the attributes of the generic model. The dependencies, which we discuss in Sect. 4, can be provided by a user with understanding of the NoSQL system's application domain or automatically using existing mining techniques. We provide a brief overview of these steps in the following sections. More detail is available in an extended technical report [14].

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 Fig. 1. This step is automated, using the procedure described in Sect. 5. Our algorithm ensures that redundancy in the physical schema captured by the provided dependencies is removed.

3 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 (e.g., a document collection). A relation schema consists of a unique relation name plus a set of attribute names. Attribute names are unique within each relation schema.

If the NoSQL database includes a well-defined schema, then describing the physical schema required for renormalization is a trivial task. The generic schema simply identifies the attributes that are present in the table, and gives names to both the attributes and the table itself. For example, Cassandra stores table definitions which can directly provide the generic schema.

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, Izquierdo et al. [7] have proposed a method for extracting a schema from JSON records in a document store, which could be applied to extract the generic physical schema required for renormalization.

4 Dependency Input

The second step of the renormalization process is to identify dependencies among attributes in the generic physical schema. Our algorithm uses two types of dependencies: functional dependencies (FDs) and inclusion dependencies (INDs). These two forms of dependencies are easy to express and are commonly used in database design [11].

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 INDs to be used to indicate foreign key relationships which exist in the denormalized data. Indeed, Mannila and Räihä [11] have previously argued that only key-based dependencies are relevant to logical design.

5 Normalization Algorithm

Levene and Vincent [9] define a normal form for database relations involving FDs and INDs 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 INDs is non-circular. 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 INDs $R(A) = S(B)$ 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 FDs and INDs while still allowing the expression of useful information such as foreign keys. As we show in Sect. 6, this produces useful logical models for a real-world example.

Data: A set of relations $\mathbf{R}$, FDs $\mathbf{F}$, and INDs $\mathbf{I}$
Result: A normalized set of relations $\mathbf{R}'''$ and new dependencies $\mathbf{F}'$ and $\mathbf{I}^{+'''}$
begin
  $\mathbf{F}', \mathbf{I}^{+} \leftarrow$ `Expand(`$\boldsymbol{F}, \boldsymbol{I}$`);` `// Perform dependency inference`
  $\mathbf{R}', \mathbf{I}^{+'} \leftarrow$ `BCNFDecompose(`$\boldsymbol{R}, \boldsymbol{F}', \boldsymbol{I}^{+}$`);` `// BCNF normalization`
  $\mathbf{R}'', \mathbf{I}^{+''} \leftarrow$ `Fold(`$\boldsymbol{R}', \boldsymbol{F}', \boldsymbol{I}^{+'}$`);` `// Remove attributes/relations`
  $\mathbf{R}''', \mathbf{I}^{+'''} \leftarrow$ `BreakCycles(`$\boldsymbol{R}'', \boldsymbol{I}^{+''}$`);` `// Break circular INDs`
end

Fig. 2. Algorithm for normalization to interaction-free IDNF

Figure 2 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 Fig. 3.

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

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

Inclusion Dependencies

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

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

Fig. 3. Example generic physical schema and dependencies.

5.1 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 [1] provides a well-known set of axioms which can be used to infer FDs from those provided as input. Similarly, Mitchell [15] 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 rules to infer new FDs for INDs and vice versa. The pullback rule enables new FDs to be inferred from FDs and INDs. The collection rule allows the inference of new INDs. These new dependencies allow the elimination of attributes and relations via the `Fold` algorithm (see Sect. 5.3) to reduce the size of the resulting schema while maintaining the same semantic information.

There is no complete axiomatization for FDs and INDs taken together [3]. Our `Expand` procedure, which uses Mitchell's pullback and collection rules for 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.

5.2 BCNF Decomposition

The second step, `BCNFDecompose`, is to perform a lossless join BCNF decomposition of the physical schema using the expanded set of FDs. 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 INDs which represent the correspondence of attributes between the decomposed relations. For example, when performing the decomposition $R(ABC) \rightarrow R'(AB), R''(BC)$ we also add the INDs $R'(B) \subseteq R''(B)$ and $R''(B) \subseteq R'(B)$. In our running example, we are left with the relations and dependencies shown in Fig. 4 after the `Expand` and `BCNFDecompose` steps.

Physical Schema

$\text{Employees}\left(\underline{\text{EmpID}}, \text{EmpName}, \text{DeptID}\right)$ $\text{Departments}\left(\underline{\text{DeptID}}, \text{DeptName}\right)$
$\text{EmpProjects}\left(\underline{\text{EmpID}}, \underline{\text{ProjID}}\right)$ $\text{EmpProjects}'\left(\underline{\text{EmpID}}, \text{EmpName}\right)$
$\text{Managers}\left(\underline{\text{DeptID}}, \text{EmpID}\right)$ $\text{Projects}\left(\underline{\text{ProjID}}, \text{ProjName}\right)$

Functional Dependencies

$\text{Employees} : \text{EmpID} \rightarrow \text{EmpName}, \text{DeptID}$ $\text{Departments} : \text{DeptID} \rightarrow \text{DeptName}$
$\text{Projects} : \text{ProjID} \rightarrow \text{ProjName}$ $\text{Managers} : \text{DeptID} \rightarrow \text{EmpID}$
$\text{EmpProjects}' : \text{EmpID} \rightarrow \text{EmpName}$

Inclusion Dependencies

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

Fig. 4. Relations and dependencies after BCNF decomposition. Note that $=$ is used to represent bidirectional inclusion dependencies.

5.3 Folding

Casanova and de Sa term the technique of removing redundant relations *folding* [2]. A complete description of our algorithm, `Fold`, is given in an extended technical report [14]. `Fold` identifies attributes or relations which are recoverable from other relations. Specifically, 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 sharing a common key.

Consider the `EmpProjects′` relation which contains the `EmpName` attribute. Since we have the IND `EmpProjects'(EmpID,EmpName)` $\subseteq$ `Employees`$(\ldots)$ and the FD `Employees: EmpID` $\rightarrow$ `EmpName` we can infer that the `EmpName` attribute in `EmpProjects′` is redundant since it can be recovered by joining with the `Employees` relation.

5.4 Breaking IND Cycles

Interaction-free IDNF requires that the final schema be free of circular INDs. Mannila and Räihä [11] use a technique, which we call `BreakCycles`, to break circular INDs when performing logical database design. We adopt this technique to break IND cycles which are not proper circular.

5.5 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 [5] with respect to **F**, (2) all the INDs in **I** are key-based or proper circular, and (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. The schema produced by the normalization algorithm of Fig. 2 is in interaction-free IDNF. We provide a proof in an extended technical report [14].

6 RUBiS Case Study

In previous work, we developed a tool called NoSE [13], which performs automated schema design for NoSQL systems. We used NoSE to generate two Cassandra schemas, each optimized for a different workload (a full description is given in an extended technical report [14]). In each case, NoSE starts with a conceptual model of the database which includes six types of entities (e.g., users, and items) and relationships between them. The two physical designs consist of 9 and 14 Cassandra column families.

Our case study uses NoSE's denormalized schemas as input to our algorithm so that we can compare the schemas that it produces with the original conceptual model. For each physical schema, we tested our algorithm with two sets of dependencies: one manually generated from the physical schema, and a second mined from an instance of that schema using techniques discussed in an extended technical report [14]. The first set of dependencies resulted in a conceptual model that was identical (aside from names of relations and attributes) to the original conceptual model used by NoSE, as desired.

For the second set of tests, renormalization produced the original model for the smaller Cassandra schema. The mining process identified 61 FDs and 314 INDs. Ranking heuristics were critical to this success. Without them, spurious dependencies lead to undesirable entities in the output schema. For the larger schema, mining found 86 FDs and 600 INDs, many of them spurious, resulting in a model different from the original. No set of heuristics will be successful in all cases and this is an area for future work.

These examples show that FDs and INDs 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 Related Work

Much of the existing work in normalization revolves around eliminating redundancy in relational tables based on different forms of dependencies. However, it does not deal with the case where applications duplicate data across relations. Inclusion dependencies are a natural way to express this duplication. Other researchers have established normal forms using inclusion dependencies [9–11] in addition to FDs. Our approach borrows from Manilla and Räihä, who present a variant of a normal form involving inclusion dependencies and an interactive normalization algorithm. However, it does not produce useful schemas in the presence of heavily denormalized data. Specifically, their approach is not able to eliminate all data duplicated in different relations.

A related set of work exists in database reverse engineering (DBRE). The goal of DBRE is to produce an understanding of the semantics of a database instance, commonly through the construction of a higher level model of the data. Unlike our work, many approaches [4,17] present only an informal process and not a specific algorithm.

There is significant existing work in automatically mining both functional [12] and inclusion [8] dependencies from both database instances and queries. These approaches complement our techniques since we can provide the mined dependencies as input into our algorithm. Papenbrock and Naumann [16] present of heuristics for making use of mined dependencies to normalize a schema according to BCNF. We leverage these to incorporate mining into our algorithm as discussed in an extended technical report [14].

8 Conclusions and Future Work

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 as application requirements change.

References

1. Armstrong, W.W.: Dependency structures of data base relationships. In: IFIP Congress, pp. 580–583 (1974)
2. Casanova, M.A., de Sa, J.E.A.: Mapping uninterpreted schemes into entity-relationship diagrams: two applications to conceptual schema design. IBM J. Res. Develop. **28**(1), 82–94 (1984)
3. Casanova, M.A.: Inclusion dependencies and their interaction with functional dependencies. J. Comput. Syst. Sci. **28**(1), 29–59 (1984)

4. Chiang, R.H., et al.: Reverse engineering of relational databases: extraction of an EER model from a relational database. Data Knowl. Eng. **12**(2), 107–142 (1994)
5. Codd, E.F.: Recent investigations into relational data base systems. Technical report RJ1385, IBM, April 1974
6. Gómez, P., Casallas, R., Roncancio, C.: Data schema does matter, even in NoSQL systems! In: RCIS 2016, June 2016
7. Cánovas Izquierdo, J.L., Cabot, J.: Discovering implicit schemas in JSON data. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 68–83. Springer, Heidelberg (2013)
8. Kantola, M., et al.: Discovering functional and inclusion dependencies in relational databases. Int. J. Intell. Syst. **7**(7), 591–607 (1992)
9. Levene, M., Vincent, M.W.: Justification for inclusion dependency normal form. IEEE TKDE **12**(2), 281–291 (2000)
10. Ling, T.W., Goh, C.H.: Logical database design with inclusion dependencies. In: ICDE 1992, pp. 642–649, Feburary 1992
11. Mannila, H., Räihä, K.J.: Inclusion dependencies in database design, pp. 713–718. IEEE Computer Society, February 1986
12. Mannila, H., Räihä, K.J.: Algorithms for inferring functional dependencies from relations. DKE **12**(1), 83–99 (1994)
13. Mior, M.J., Salem, K., Aboulnaga, A., Liu, R.: NoSE: Schema design for NoSQL applications. In: ICDE 2016, pp. 181–192 (2016)
14. Mior, M.J., Salem, K.: Renormalization of NoSQL database schemas. Technical report CS-2017-02, University of Waterloo (2017)
15. Mitchell, J.C.: Inference rules for functional and inclusion dependencies. In: PODS 1983, pp. 58–69. ACM (1983)
16. Papenbrock, T., Naumann, F.: Data-driven schema normalization. In: Proceedings of EDBT 2017, pp. 342–353 (2017)
17. Premerlani, W.J., Blaha, M.R.: An approach for reverse engineering of relational databases. Commun. ACM **37**(5), 42–49 (1994)