Data & Knowledge Engineering 142 (2022) 102089

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

A workload-driven method for designing aggregate-oriented NoSQL databases

Liu Chen [a,b], Ali Davoudian [c], Mengchi Liu [d,*]

[a] School of Computer Science, Wuhan University, Wuhan, Hubei, China
[b] School of Computer Science, Carleton University, Ottawa, Ontario, Canada
[c] IT Branch, Canada Revenue Agency, Ottawa, Ontario, Canada
[d] School of Computer Science, South China Normal University, Guangzhou, Guangdong, China

ARTICLE INFO

Keywords:
NoSQL database schemas
Database design
Workload-driven
Design trade-offs

ABSTRACT

Due to the scalability and availability problems with traditional relational database systems, a variety of NoSQL stores have emerged over the last decade to deal with big data. How data are structured in a NoSQL store has a large impact on the query and update performance and the storage usage. Thus, different from the traditional database design, not only the data structure but also the data access patterns need to be considered in the design of NoSQL database schemas. In this paper, we present a general workload-driven method for designing key-value, wide-column, and document NoSQL database schemas. We first present a generic logical model Query Path Graph (QPG) that can represent the data structures of the UML class diagram. We also define mappings from the SQL-based query patterns to QPG and from QPG to aggregate-oriented NoSQL schemas. We use a cost model to measure the query and update performance and optimize the QPG schemas. We evaluate the proposed method with several typical case studies by simulating workloads on databases with different schema designs. The results demonstrate that our method preserves the generality and the quality of the design.

1. Introduction

Emerging big data systems [1–3] aim at managing voluminous data spread between multiple servers and ensuring the low response time of queries. Such requirements for managing and querying big data have revealed the limitations of relational databases [4,5]. A new kind of non-relational database management systems, named NoSQL stores, have been developed to deal with such limitations [6–8]. They increase scalability and availability by loosening the ACID (Atomicity, Consistency, Isolation and Durability) constraints [9,10] and providing less restrictive properties, such as eventual consistency [11,12]. NoSQL stores utilize different data models for different purposes, including (among others) key-value, wide-column and document models. Despite their differences, the three data models are *aggregate-oriented* [7]. In more detail, they organize data as units of related key-value pairs in a nested way. Such a nested data unit represents data that are accessed (read/write) together in one operation [10,13]. Hence, they can facilitate costly join operations by embedding data [14,15]. This means denormalizing *1:N* or *M:N* relationships by clustering related entities, which results in data duplication and violates the traditional database design philosophy.

To tap into the benefits of data nesting or denormalization in designing the schemas of aggregate-oriented NoSQL databases, designers should not only consider what data will be stored in the database but also how such data will be accessed [15]. Despite existing well-established design methods of traditional database schemas, where even some are workload-driven [16–19], they do not fit NoSQL stores. This is mostly because they aim to achieve other targets such as ACID properties [20].

[*] Corresponding author.
E-mail addresses: dollychan@whu.edu.cn (L. Chen), ali.davoudian@cra-arc.gc.ca (A. Davoudian), liumengchi@scnu.edu.cn (M. Liu).

https://doi.org/10.1016/j.datak.2022.102089
Received 2 November 2020; Received in revised form 28 July 2022; Accepted 5 October 2022
Available online 15 October 2022

There is no well-defined methods for NoSQL database design due to their heterogeneity [21]. Data modeling techniques need to reflect different data access, different data consistency and durability requirements, different data model and query capability of NoSQL models. Several generic metamodels [22,23] have been proposed to represent both the relational model and NoSQL models with the mapping rules between the intermediate model and database models provided. However, they did not cover the design of NoSQL databases. Several methods [24–31,31–33] have been proposed for the design of aggregate-oriented NoSQL database schemas in the last several years. Most of methods [25–27,29–31] focus on a specific NoSQL data model, which results in extra efforts for designers who work with different NoSQL data models and need to change between design methods. For example, a designer who is either developing a multi-model NoSQL database [34], or migrating an old NoSQL schema to a new one [35] has to specify the same conceptual schema and query workload in different notations. On the other hand, methods that support multiple NoSQL data models do not consider how data will be accessed and updated [28,32,36]. Also, to gain a good trades-off between storage usage and query performance, a cost model is required to select a target schema in good quality, otherwise, developers need to make choices.

To deal with the above limitations, we present a workload-driven method for designing aggregate-oriented NoSQL database schemas. Our main contributions are as follows: (1) a generic logical model for aggregate-oriented NoSQL stores, (2) a set of mapping rules that guide a logical data modeling process with respect to the specific features of different NoSQL data models, as well as the conceptual data model and data access patterns, (3) a cost model to optimize the schema design by making trade-offs between the query performance and storage overhead or consistency maintenance and (4) a tool that automates the generation of aggregate-oriented database schemas according to the proposed method. It generates some scripts which instantiate the resulted database schemas, with respect to targeted NoSQL stores. We evaluate our approach with case-studies used in the NoSQL literature and generate database schemas for the popular representatives of key-value, wide-column and document stores, i.e., Oracle NoSQL [37], Cassandra [38] and MongoDB [39] respectively. As how data are structured in a NoSQL database has a large impact on query and update performance, we take state-of-the-art methods [26,27] as benchmarks and compare the generated database schema with the ones they produce. The experiments verify the quality of the generated schemas and the validity of the cost model. In some cases, our method performs even better.

The rest of the paper is organized as follows. Section 2 provides a running example and some background on the Oracle NoSQL, Cassandra and MongoDB data models. Related works are discussed in Section 3. In Section 4, we detail the different phases in our NoSQL design process. We evaluate the method in Section 5. Finally, we conclude the paper in Section 6.

2. NoSQL database models

In this section, we provide a running example to illustrate various concepts that appeared in this paper. We also introduce key-value, wide-column and document data models, and their popular representatives: Oracle NoSQL [37], Cassandra [38] and MongoDB, as well as the formal definitions of their corresponding database schemas.

2.1. Running example

We use the sample database for a real e-commerce system from de Lima and Mello [27]. Fig. 1 shows the conceptual schema of this system in the UML class diagram notations. In this online store, customers can request orders consisting of different products. Each order has payment information and the carrier that delivers it. A product belongs to a category and has a supplier.

The following are query patterns used in this online shopping application:

- **Q_1**. Given an *order id*, return the *order* and related *customer*, *items* and *products*.
- **Q_2**. Given an *order id*, return the *order* and related *customer* and *payments*.
- **Q_3**. Given a *customer id*, return all *orders* and related *carriers*.
- **Q_4**. Given a *customer id*, return all *orders* and related *payments*.
- **Q_5**. Given a *product id*, return all related *items* and *orders*.
- **Q_6**. Given a *supplier id*, return the *supplier* and all related *products* including their *categories*.

Besides the query patterns, inserting/updating operations over single class or relationship are considered as write patterns. For example, writes on a customer are adding or modifying its attributes, and writes on a relationship $requests$ are creating or deleting the connection between the primary key of a customer and the primary key of an order.

2.2. Key-value stores

Basic key-value stores organize data as a simple collection of *Key-Value* (KV) pairs, where the *Value* part is uniquely identified by a simple indexed *Key* part. Such schema-less stores encode values as byte arrays (e.g., BLOB), where their serialization/deserialization is left to the client application. Thus, indexing and querying based on the *Value* part are not supported by such systems.

Oracle NoSQL is an advanced KV store where the *Key* part is structured as the combination of a major key and an optional minor key separated by the hyphen ('–'). Both keys can be simple or composite with a sequence of components, where the components are separated by the forward-slash ('/'). The *Value* part can be either schema-less as a byte array or schema-full using Table API [40]. In a distributed system, KV pairs are spread evenly across partitions through hashing their major keys. Applications can take advantage of major keys to achieve data locality for all KV pairs that share the same major key. More precisely, the proper design of major

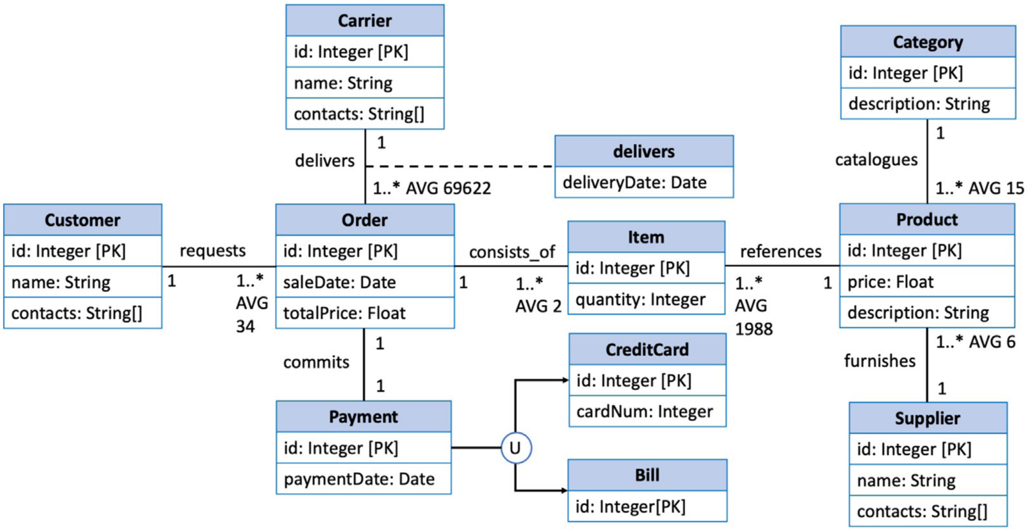


Fig. 1. Conceptual schema of an e-commerce platform.

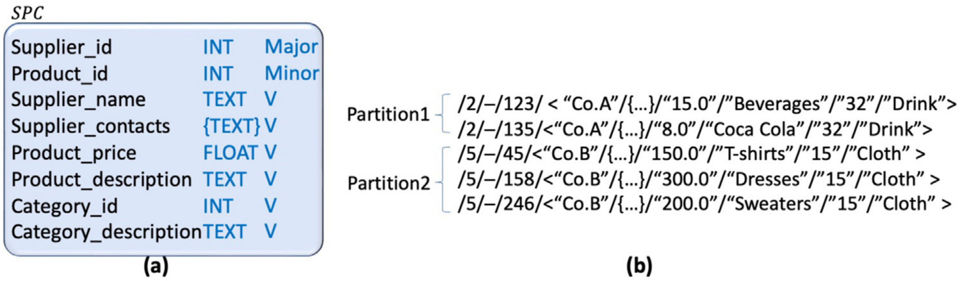

SPC
Supplier_id INT Major
Product_id INT Minor
Supplier_name TEXT V
Supplier_contacts {TEXT} V
Product_price FLOAT V
Product_description TEXT V
Category_id INT V
Category_description TEXT V
(a)
Partition1
/2/–/123/ < "Co.A"/{...}/"15.0"/"Beverages"/"32"/"Drink">
/2/–/135/<"Co.A"/{...}/"8.0"/"Coca Cola"/"32"/"Drink">
Partition2
/5/–/45/<"Co.B"/{...}/"150.0"/"T-shirts"/"15"/"Cloth" >
/5/–/158/<"Co.B"/{...}/"300.0"/"Dresses"/"15"/"Cloth" >
/5/–/246/<"Co.B"/{...}/"200.0"/"Sweaters"/"15"/"Cloth" >
(b)

Fig. 2. (a) a KV logical schema that satisfies the query Q_6 in our running example; and (b) some instances of the schema.

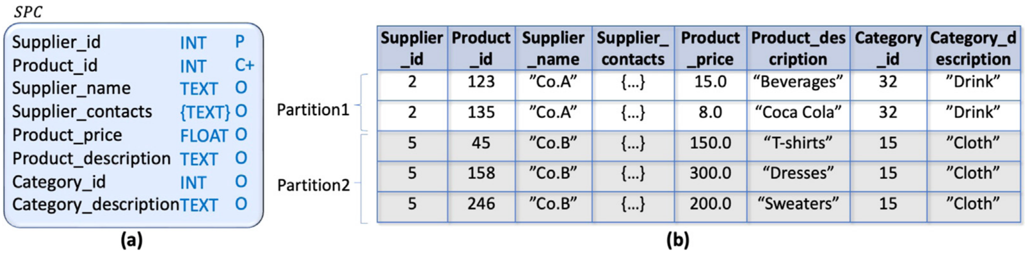
keys results in applying a single atomic operation with ACID guarantees on multiple KV pairs that share the same major key, and the even distribution of KV pairs across partitions. Also, KV pairs that share the same major key are sorted in ascending order of the value of the first component in the sequence of minor key components. An Oracle NoSQL database schema is a set of KV schemas, defined as follows.

Definition 1. A KV schema is a sequence of components divided into the major key, the minor key and the value part, denoted by $S(c_1, c_2, \ldots, c_n)$, where S is the schema name, and each component c_i is associated with a type t_i and a role r_i, t_i is either a primitive INT, FLOAT, TEXT or DATE, or a collection of primitives constructed with a set ({}), a list ([ ]), or a map[1] (<>) constructor, and r_i specifies that the component is a part of either the major key (MAJOR), the minor key (MINOR), or values (V). There are at least one MAJOR and one V components.

Oracle NoSQL provides simple atomic operations to access and modify individual KV pairs including *put(key, value)* to add or modify a pair and *get(Key)* to retrieve a value corresponding to the *key*. It also provides an atomic *multiGet(parentKey)* operation to return multiple sorted KV pairs having the same *parentKey*. Note that *parentKey* is a subsequence of the major key components. Moreover, it offers an execute operation to execute multiple put operations in an atomic and efficient way (provided that the keys specified in these operations all share the same major key).

For example, Fig. 2(a) shows a KV schema named *SPC*, and Fig. 2(b) shows some instances of the schema where *Supplier_id*, as the major key, along with *Product_id*, as the minor key, together ensure the uniqueness of each pair. The major key *Supplier_id* ensures that all products provided by a supplier are hosted in the same partition. The minor key *Product_id* ensures that all products associated with the same supplier are sorted based on their ids. As shown in the figure, the first instance is composed of "2"/–/"123" as its *Key*, and a byte array of the associated product and category as the *Value*. With the above schema design, the query Q_6 can be answered by a single read request *get(Supplier_id)*.

[1] It is a container of key-value pairs.



SPC

Supplier_id	INT	P
Product_id	INT	C+
Supplier_name	TEXT	O
Supplier_contacts	{TEXT}	O
Product_price	FLOAT	O
Product_description	TEXT	O
Category_id	INT	O
Category_description	TEXT	O

(a)

	Supplier_id	Product_id	Supplier_name	Supplier_contacts	Product_price	Product_description	Category_id	Category_description
Partition1	2	123	”Co.A”	{...}	15.0	“Beverages”	32	”Drink”
	2	135	”Co.A”	{...}	8.0	“Coca Cola”	32	”Drink”
Partition2	5	45	”Co.B”	{...}	150.0	“T-shirts”	15	”Cloth”
	5	158	”Co.B”	{...}	300.0	“Dresses”	15	”Cloth”
	5	246	”Co.B”	{...}	200.0	“Sweaters”	15	”Cloth”

(b)

Fig. 3. (a) A CF logical schema that satisfies the queries Q_6 in our running example; and (b) the corresponding instance.

2.3. Wide-column stores

In wide-column stores, data are organized in tables with flexible schemas. A table is a collection of rows, where each row contains a flexible number of columns, and each column is associated with a name and a value. Each row is uniquely identified by a row key. Correlated columns are grouped in a Column Family (CF) and then saved together.

In some wide-column stores such as Google BigTable [41] and HBase [42], rows are sorted by the row keys and related data are stored in contiguous rows. A row key can include multiple values separated by a delimiter. In other wide-column stores such as Cassandra, a row key is comprised of a mandatory partition key and an optional clustering key. Both keys can be simple as one column or composite as a sequence of columns. Cassandra hashes the value of the partition key in order to determine which node to store the corresponding rows, which is different from the row key based range partitioning used by Google Bigtable or HBase. In this way, rows with the same partition key are stored together to increase the locality of accesses. The clustering key is used to sort rows inside each partition. In this paper, we use Cassandra as the target wide-column store.

The wide-column database schema is a set of CF schemas, defined as follows.

Definition 2. A CF schema is a sequence of components divided into the partition key, the clustering key and ordinary components, denoted by $S(c_1, c_2, \dots, c_n)$, where S is the schema name, and each component c_i corresponds to a type t_i and a role r_i, t_i is either a primitive INT, FLOAT, TEXT or DATE, or a collection of constructed types using set “{}”, list “[ ]”, or map “<>” constructors on primitives; and r_i is either a component of the Partition key specified by P, the Clustering key specified by C [+|−] for ascending or descending order, or an Ordinary column specified by O. There are at least one P and one O components.

Data access operations are performed on single CF. Given the primary key that consists of both the partition key and the clustering key, a unique row can be fetched or written. Given the partition key and optional predicates on clustering key columns, multiple rows can be fetched. Additionally, the rows found can be ordered based on the clustering key.

For example, Fig. 3(a) and (b) show a CF logical schema named *SPC* designed for query Q_6 in our running example, and the corresponding instance, respectively. Here, *Supplier_id*, as a single component of the partition key, along with *Product_id*, as the clustering key, constitute the primary key of the column family. Each row stores a product and the related supplier and category. The selection of *Supplier_id* as the partition key ensures that all products associated with a supplier are stored in the same partition. Thus, all data needed by query Q_6 can be fetched in one look-up with the given partition key *Supplier_id* of the CF SPC.

2.4. Document stores

Document stores are extended key-value stores in which the value is represented as a document. They organize data as Document Collections (DCs). A Document Collection (DC) is identified by its name and consists of a set of documents representing the same category of information. A document has a set of properties (or key-value pairs) that can be either primitive (e.g. integer, string) or an embedded document. As document stores know the format of documents, they allow efficient retrievals via indices and search functions over the keys and values of properties. Documents in MongoDB are stored in BSON format [43], which is a JSON-like structure. It allows the structure of a DC schema to be defined via a JSON schema [44]. A document collection can be partitioned, and documents are distributed to shards based on the shard key, which is one or more document properties. MongoDB adopts range partitioning based on the original value or the hashed value of the shard key. A MongoDB database schema is a set of DC schemas, defined as follows.

Definition 3. A DC schema is a set of hierarchical properties, denoted by $S(p_1, p_2, \dots, p_n)$, where S is the schema name, and each property p_i corresponds to a type t_i and a role r_i, t_i is either a primitive INT, FLOAT, TEXT or DATE, an embedded document specified by D, a reference to a document specified by *n, where n is the name of the document collection, or a collection of constructed type using set “{}”, list “[ ]”, and map “<>” constructors on either primitives, documents or references to documents; and r_i is either a component of the shard key specified by K, or a regular property specified by NULL.

Data access operations are usually over individual documents, which are units of data distribution and data manipulation. The basic operations provided by MongoDB are as follows: $insert(col, doc)$ adds a document doc into collection col; and $find(col, selector)$

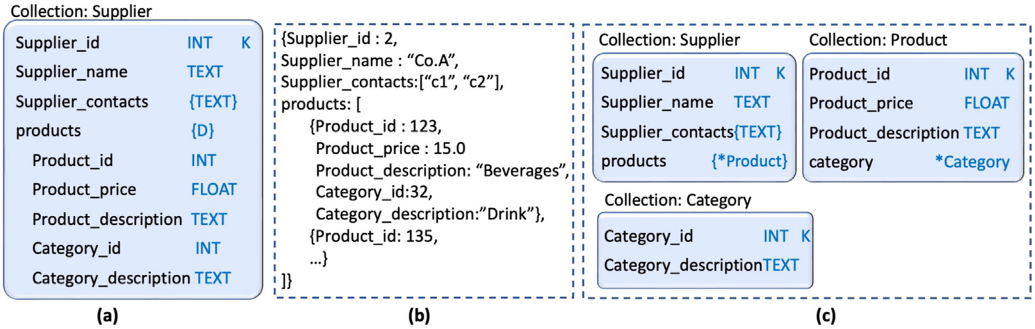


Fig. 4. Two DC schemas that satisfy Q_6 in our running example, by either (a) embedded documents and (b) sample document of (a) or (c) referenced documents.

retrieves from collection col all documents matching the $selector$ by performing a collection scan. The simplest selector is the empty document $\{\}$, which matches every document so that all documents in col are retrieved. The selector follows the hierarchical structure of the document, and embedded documents can be accessed by specifying paths, for example, "$products.product\_id$" can be used to access the products embedded in the supplier of the sample document in Fig. 4(b).

Fig. 4(a) and (b) show a logical schema and a sample for the DC schema named *Collection:Supplier* that embeds products and associated categories with each supplier in our running example, where *Supplier_id* is the shard key. This allows an efficient retrieval of information in query Q_6 in Fig. 4(a) by using $find(Supplier, \{Supplier\_id\ :?\})$ without using the costly join operations. More precisely, each time a supplier is retrieved, data about its associated products and related categories are also returned, which avoids having to perform joins with data contained in another document collection. On the other hand, as Fig. 4(c) shows, we may prefer to join suppliers, products and categories rather than embedding because of the trade-off between the query and update performance.

3. Related works

In this section, we briefly describe the existing works that address the design of NoSQL database schemas. All related works are summarized in Table 1.

NoSQL stores are known as schema-less when storing data, but it requires the schema in order to process the data; that is, "schema on read". Reverse engineering methods [48–52] are proposed to extract schemas from the physical NoSQL databases. Abdelhedi et al. [49,50] first extract the physical model from MongoDB and then generate the conceptual model in the UML class diagram. Molina et al. [51,52] propose a data model to represent the variants of properties belonging to an entity type that can be extracted from a document database, and then generate the schema required by the Object-Document Mappers to convert objects into document databases for object-oriented applications.

Some works [25,27,29,30] focus on specific NoSQL stores. Li [29] proposes one of the very first work on the schema design of NoSQL databases. It provides a set of heuristics, whereby the relational schema is mapped into the HBase database schema [42]. Chebotko et al. [25] propose a workload-driven design method for transforming an Entity-Relationship (ER) model into a database schema in Cassandra. This mapping is based on a set of rules regarding the application workflow describing query or data access patterns. As this mapping generates a CF for each query, it might end up with a high level of denormalization that, in turn, increases the performance of read queries. However, as the cost of consistency maintenance is increased, the performance of write queries might be decreased. UMLtoGraphDB [46] proposes a metamodel for Graph databases and supports transforming the UML class diagram into the graph model. It also uses Object Constraint Language (OCL) to define business rules, queries and invariants and provides mappings from OCL constraints to graph query language Gremlin. However, the OCL constraints are not considered when generating the graph model. de Lima and Mello [27] propose a similar design method, but focusing on MongoDB. It converts an Extended-Entity-Relationship (EER) model into a logical model to represent collections of documents. This transformation is based on a set of rules regarding the workload information of estimated query frequencies and volume of data, in order to avoid data redundancy. Jia et al. [30] propose model transformation and data migration methods from relational database to MongoDB. It takes the query and data statistics extracted from the relational database into consideration, and supports automatic transformation of relational database in ER model to documents in MongoDB. Documents are embedded if a relationship is labeled with "Frequent Join" tag, and the corresponding entity types are not labeled with "Frequent Update", "Frequent Insert" or "Big Size" tags.

Some works focus on generic logical metamodels so that transformations between multiple database models can be supported with the mapping rules provided [22,23,28,32,36,53]. SQLtoKeyNoSQL [22] uses a hierarchical canonical model as an intermediate model between relational schema and key-based NoSQL data models (i.e., KV, CF and DC). U-Schema [23] is a unified metamodel to cover all data model concepts of the relational schema and four NoSQL logical schemas (i.e., KV, CF, DC, and graph). SQLtoKeyNoSQL [22] and U-schema [23] do not cover a conceptual model.

UMLtoNoSQL [32,53] presents an approach to automatically translate conceptual models expressed in UML class diagrams to several NoSQL database models including Cassandra (CF), MongoDB (DC) and Neo4j (graph). It proposes a generic logical metamodel, which consists of a set of tables and a set of binary relationships between tables. Tables and relationships correspond

Table 1
Database schema design processes for aggregate-oriented NoSQL stores.

	Conceptual schema	Logical schema	Supported data models	Queries	Workload information	Process automation
Li [29]	✗	Relational	W	✗	✗	✗
Chebotko et al. [25]	ER	In-house notation for CFs	W	ERQL [45]	✗	✓
UMLtoGraphDB [46]	UML	A metamodel for GraphDB	G	OCL constraints	✗	✓
de Lima and Mello [27]	EER	In-house notation for documents	D	XML-based [47]	Estimated query frequencies & data volume	✓
Jia et al. [30]	ER	Relational	D	SQL	Frequent join & big size & frequent modify/insert	✓
SQLtoKeyNoSQL [22]	✗	Hierarchical canonical model	R,K,W,D	✗	✗	✓
Candel et al. [23]	✗	U-Schema	R,K,W,D,G	✗	✗	✓
UMLtoNoSQL [32]	UML	Generic model for NoSQL	W,D,G	✗	✗	✓
ModelDrivenGuide [36]	UML	Physical independent metamodel	R,K,W,D,G	✗	✗	✗
Atzeni et al. [28]	UML	NoAM	K,W,D	✗	✗	✗
Mortadelo [33]	UML-like	Generic data metamodel	W,D	SQL-based	Highly update	✓
Mior et al. [26]	ER	✗	W	SQL-based	Estimated query frequencies & data volume	✓
DBSR [31]	ER	Directed acyclic graph	D	Binary join	✗	✓
Our method	UML	Query path graph	K,W,D	SQL-based	Estimated query/update frequencies & data volume	✓

Key-value (**K**), Wide-column (**W**), Document (**D**), Graph (**G**), Relational (**R**).

to classes and relationships in the UML class diagram. Every table is then mapped to a column family/document/graph node in the physical model. Several mapping rules for relationships are provided and candidate solutions for mapping a relationship are all generated. ModelDrivenGuide [36] proposes a physical independent method for mapping a UML class diagram to both the relational and NoSQL models (i.e., KV, CF, DC and graph). It uses a common logical metamodel to integrate all the concepts in five physical models, and provides the transformation and refinement rules between and inside metamodels. A set of heuristics is provided to remove ineffective solutions. UMLtoNoSQL and ModelDrivenGuide are workload-agnostic so that they do not generate database schemas that can efficiently implement any particular workload.

Atzeni et al. [28] propose a logical model called NoSQL Abstract Model (NoAM) that organizes data in a set of collections, and each collection is a set of blocks identified by block keys. A block has more than one entry, and each entry is a key-value pair where the value can be scalar or another block. It groups related entities into aggregates based on the data access patterns, scalability and consistency requirement. Aggregates are then transformed into NoAM blocks by a number of guidelines. However, the process to determine the aggregates from a UML class diagram to NoAM blocks lacks formalizations and algorithms to achieve automation. Also, the NoAM block is an abstraction of NoSQL constructs such as CFs or DCs, and it may be problematic in identifying important features of the target data model, such as the partition and clustering keys of a CF schema.

Mortadelo [33] proposes a Generic Data Metamodel (GDM) for conceptual model and two NoSQL data models (i.e., CF and DC) as well as the query workload. Highly Used (HU) entity types are used to control the denormalization levels of entities. However, there is no cost model used to evaluation the validity for normalizing HU entities.

Some works [26,31] focus on schema evaluation among all possible solutions to select a suitable target physical schema based on a cost model. Mior et al. [26] propose a tool, called NoSQL Schema Evaluator (NoSE), which takes into account all alternative plans to generate a wide-column database schema. It maps a given ER model along with the workload information of estimated query frequencies and volume of data into all candidate plans and selects the one offering the best performance through a Binary Integer Programming (BIP) approach [54]. DBSR [31] takes the conceptual model and read workload as input, and generates a suggested document schema and the corresponding query plan recommendations. It takes a bottom-up approach; that is, each entity in ER model is firstly mapped to a document, then all possible optimizations are iterated, meanwhile either embedded documents or referenced documents are built for relationships. However, it only considers the read patterns of binary joins, and other query operations and update patterns are not covered.

4. Our approach

Fig. 5 shows how our method is used to design aggregate-oriented NoSQL database schemas. We define a generic logical NoSQL model called Query Path Graph (QPG) that can reflect the data structure, data access pattern as well as the query capabilities,

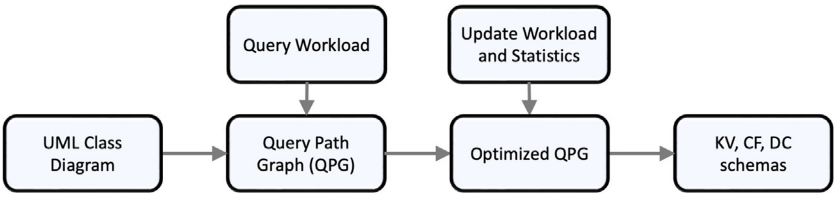


Fig. 5. An overview of our method for designing NoSQL database schemas.

partitioning and scalability features of aggregate-oriented NoSQL stores. Our method first generates an initial QPG schema from the conceptual model represented by a UML class diagram and a set of query patterns. Based on class cardinalities, relationship multiplicities, and query/update frequencies, we use a cost model to optimize the QPG, which makes a trade-off between query and update costs. Finally, the optimized QPG is mapped into the KV, CF and DC schemas. The details are explained in the following sections.

4.1. Conceptual schema

In order to create a NoSQL database schema, as discussed in the earlier works [25–28], a conceptual schema is usually used to define a succinct overview of the data requirements and provide detailed descriptions of the entities, relationships, and constraints [55]. The conceptual schema reflects the application requirement and workflow, and describes the domain of interests. We represent this schema using the most fundamental and widely used UML class diagram with the following features.

- A class has a name and a number of attributes, and the primary key of the class consists of one or more attributes.
- A relationship has a name and a number of attributes, as well as multiplicities indicating how many objects of one class can be associated with objects of another class, which are represented as 1, *, 0..1, 1..* or 0..*, where 1 and * indicate 1 or more respectively and the rest represent the ranges.

Fig. 1 shows the conceptual schema that has eight classes: *Customer*, *Order*, *Item*, *Product*, *Category*, *Supplier*, *Payment* and *Carrier*, as well as their attributes labeled $[PK]$ to indicate they are part of the primary key, and relationships with multiplicities and attributes.

4.2. Workload modeling

An application workload is a set of query/update patterns, represented as $W = Q \cup \mathcal{U}$, where Q is a set of query patterns and $\mathcal{U} = C \cup R$ is a set of update patterns consisting of a set of classes and relationships. Similar to GMAP [56] with a SQL-based query language independent of database schemas, a SQL-based language with the following simple syntax is used to represent query patterns over the UML class diagram:

```
SELECT attributes
FROM query path (, query path)*
WHERE attribute {= | < | <= | > | >=} value
[AND ...]
[ORDER BY attribute[(ASC)]|[(DESC)] [, ...]]
```

where the FROM clause specifies *query paths* with classes and relationships, the SELECT clause specifies the projection of attributes of one or more classes, the WHERE clause specifies one or more equality and/or inequality conditions over the classes, and the optional ORDER BY clause specifies the attributes used to order the results. Note that all kinds of binary relationships in the UML class diagram can be used in the *query paths* to represent the data needed by a query pattern, and the classes and relationships used in the *query paths* form a subgraph of the UML class diagram. Aggregate-oriented NoSQL stores support simple queries and allow data to be aggregated in a hierarchical structure as introduced in Section 2. We assume that the *query paths* form an acyclic graph, as the existence of a cycle would result in infinite loops when nesting data.

The following statement shows the SQL-based query for Q_6 in the running example, which finds supplier, product and category details with a given specific supplier:

```
SELECT Supplier.id, Supplier.name, Supplier.contacts, Product.id,
  Product.price,Product.description,Category.id,Category.description
FROM Supplier.furnishes.Product.catalogues.Category
WHERE Supplier.id = ?;
```

The query uses objects in classes $Supplier$, $Product$ and $Category$ as well as the relationships between them and returns the results that match the given $Supplier.id$. Fig. 6 shows the *query paths* of the query patterns presented in Section 2.1. Note that query patterns Q_2 and Q_4 use the union type $Payment$. Since the *query paths* indicate classes and corresponding relationships required by the query, relationships of special types such as generalization or union type are treated as regular one-to-one binary relationships.

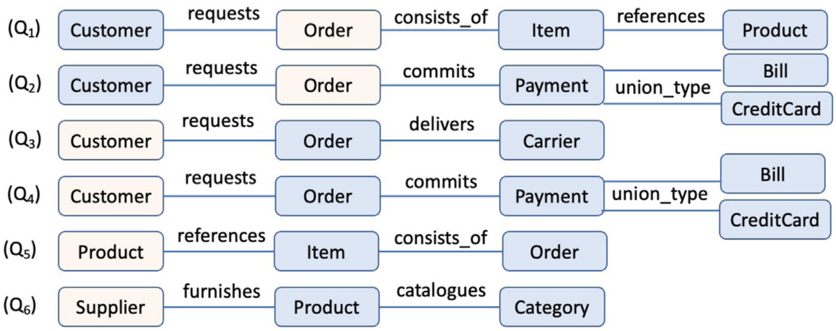

(Q1) Customer requests Order consists_of Item references Product
(Q2) Customer requests Order commits Payment union_type Bill CreditCard
(Q3) Customer requests Order delivers Carrier
(Q4) Customer requests Order commits Payment union_type Bill CreditCard
(Q5) Product references Item consists_of Order
(Q6) Supplier furnishes Product catalogues Category

Fig. 6. The query paths of query patterns of the running example in Section 2.1.

4.3. Query path graph

A conceptual schema specifies what data structures the database is composed of. But it is not enough to design a NoSQL database schema as how data are accessed is not considered [25–27]. We define a generic NoSQL logical model called Query Path Graph (QPG). The goal of QPG is to represent both the data structures and the data access patterns by aggregation trees, and each aggregation tree can be mapped to a NoSQL schema (i.e. KV, CF or DC) in the end. The definitions of Query Path Graph (QPG), Aggregation Tree and Execution Plan are given as follows.

Definition 4. Given a UML class diagram $D = (C, R)$, where $C = \{c_1, \dots, c_n\}$ is a set of classes and $R = \{(c_i, c_j)|c_i, c_j \in C\}$ is a set of relationships. A relationship $r = (c_i, c_j)$ represents the mutual connection between class c_i and c_j. Both class and relationship have a name and a set of attributes. A Query Path Graph (QPG) is defined as a labeled directed multigraph $G = (N, E, L)$, where

(1) Every class in C is a node in N. Besides having name and attributes, each attribute may have a set of Scalar Attribute (SA) specifications of four kinds: Equality Attribute (EA), Inequality Attribute (IA), Ordering Attribute (OA) and Projection Attribute (PA). Each SA specification is a pair of SA type and a set of aggregation labels, which indicates the functionalities of the attribute in aggregation trees defined in Definition 5. Also, a node can be assigned as the Access Point $[AP]$ of one or more aggregation trees.

(2) $E \subseteq N \times N$ is a set of directed edges. Each directed edge $e = (n_i, n_j) \in E$ corresponds to a relationship $r = (c_i, c_j) \in R$. Besides having name, attributes and multiplicities on relationships, an edge is also associated with a set of aggregation labels indicating one or more aggregation trees the edge belongs to, see Definition 5. Similar to attributes of QPG nodes, an attribute of QPG edges has a set of SA specifications as well.

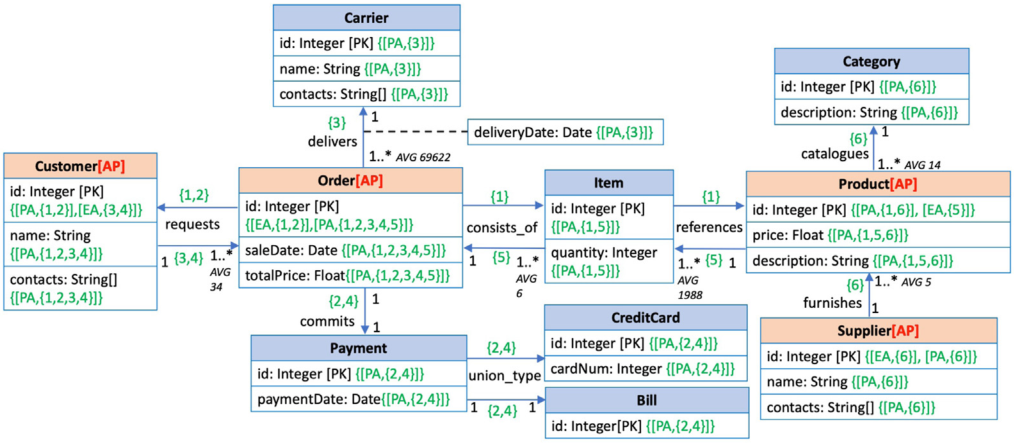
(3) L is a set of aggregation labels and each of them indicates an aggregation tree $G(l)$, with $l \in L$.

Definition 5. An aggregation tree $G(l) = (N', E', \{l\})$ is a sub-graph of $G = (N, E, L)$, where $l \in L$; $E' \subseteq E$ and for each $e \in E'$, the set of aggregation labels associated with e contains l; and $N' \subseteq N$ is the union of all nodes of edges in E'. The root of the tree is the only node in N' that does not have any incoming edge.

Fig. 7(a) shows an example of the QPG schema, which consists of eight nodes, ten directed edges and a set of aggregation labels indicating six aggregation trees. The aggregation tree labeled 6 consists of nodes *Supplier*, *Product* and *Category*, and edges *furnishes* and *catalogues* with node *Supplier* as the root. As an edge may participate in more than one aggregation tree, there is a set of labels associated with an edge. The edge $commits$ is associated with $\{2, 4\}$ and belongs to two aggregation trees. Accordingly, attributes of nodes and edges are also involved in one or more aggregation trees. The SA specification $\{[PA : \{1, 6\}], [EA : \{5\}]\}$ on attribute id of node $Product$ indicates that in aggregation tree labeled 5, id is used for the equality search; and in aggregation trees labeled 1 and 6 respectively, id is the attribute to be projected. More details are explained in Section 4.4.

Definition 6. Given a workload W, a QPG schema $G = (N, E, L)$ and $\mathcal{P}(L)$ the powerset of L, the execution plan Ω for the workload W on the QPG schema G is a set of mapping from W to $\mathcal{P}(L)$.

Fig. 7(b) shows the execution plan Ω for all query and update patterns based on the given QPG schema, where the first six mappings are for query patterns and the rest for update patterns on each class/relationship. Each query/update pattern is mapped to a set of aggregation labels. For example, $Q_1 \rightarrow \{1\}$ indicates that query pattern Q_1 is answered by the aggregation tree labeled 1; and $customer \rightarrow \{1, 2, 3, 4\}$ indicates that updates on objects of class $Customer$ need to access four aggregation trees labeled 1, 2, 3 and 4 respectively.


Carrier
id: Integer [PK] {[PA,{3}]}
name: String {[PA,{3}]}
contacts: String[] {[PA,{3}]}
{3}
delivers
1
1..* AVG 69622
deliveryDate: Date {[PA,{3}]}
Category
id: Integer [PK] {[PA,{6}]}
description: String {[PA,{6}]}
{6}
catalogues
1
1..* AVG 14
Customer[AP]
id: Integer [PK]
{[PA,{1,2}],[EA,{3,4}]}
name: String
{[PA,{1,2,3,4}]}
contacts: String[]
{[PA,{1,2,3,4}]}
{1,2}
requests
1 {3,4} 1..*
AVG
34
Order[AP]
id: Integer [PK]
{[EA,{1,2}],[PA,{1,2,3,4,5}]}
saleDate: Date {[PA,{1,2,3,4,5}]}
totalPrice: Float{[PA,{1,2,3,4,5}]}
{1}
consists_of
1 {5} 1..*
AVG
6
Item
id: Integer [PK]
{[PA,{1,5}]}
quantity: Integer
{[PA,{1,5}]}
{1}
references
1..* {5} 1
AVG
1988
Product[AP]
id: Integer [PK] {[PA,{1,6}], [EA,{5}]}
price: Float {[PA,{1,5,6}]}
description: String {[PA,{1,5,6}]}
{6}
furnishes
1..* AVG 5
1
{2,4}
commits
1
1
Payment
id: Integer [PK] {[PA,{2,4}]}
paymentDate: Date{[PA,{2,4}]}
{2,4}
union_type
1 {2,4} 1
CreditCard
id: Integer [PK] {[PA,{2,4}]}
cardNum: Integer {[PA,{2,4}]}
Bill
id: Integer[PK] {[PA,{2,4}]}
Supplier[AP]
id: Integer [PK] {[EA,{6}], [PA,{6}]}
name: String {[PA,{6}]}
contacts: String[] {[PA,{6}]}

(a) A QPG schema

Ω: { Q_1→{1}, Q_2→{2}, Q_3→{3}, Q_4→{4}, Q_5→{5}, Q_6→{6}, customer→{1,2,3,4}, order→{1,2,3,4,5}, item→{1,5}, Bill→{2,4}, carrier→{3}, payment→{2,4}, CreditCard→{2,4}, product→{1,5,6}, category→{6}, supplier→{6}, delivers→{3}, requests→{1,2,3,4}, commits→{2,4}, consists_of→{1,5}, references→{1,5}, catalogues→{6}, furnishes→{6}, union_type→{2,4}}

(b) The Execution Plan

Fig. 7. The initial QPG schema and the execution plan generated for the running example in Fig. 1.

4.4. Mapping Query Patterns to QPG schemas

Given a UML class diagram $D = (C, R)$ and an application workload $W = \mathcal{Q} \cup \mathcal{U}$, first the initial QPG schema $G = (N, E, L)$ and the corresponding execution plan Ω is generated as follows.

1. *Mapping UML classes to QPG nodes.*

For each class $c_i \in C$, a node n_i with all the attributes and keys in c_i is added to N. As Fig. 7 shows, the generated QPG schema has eight nodes as the classes in the UML class diagram.

2. *Assigning a QPG node as the Access Point (AP) for each query pattern $Q_k \in \mathcal{Q}$.*

NoSQL stores usually support simple queries in which all related data aggregated within an entry point can be fetched by one read; that is, classes and relationships accessed by the query pattern Q_k are connected and grouped by the attributes used in the equality search of Q_k. Thus, a node n is determined as the AP for Q_k when one or more attributes of the corresponding class $c \in C$ are used as equality predicates of Q_k. A node can be assigned as the access point for several query patterns, for example *Customer* is the access point for Q_3 and Q_4 in Fig. 7. If multiple classes have equality attributes, the one with the least total number of duplication of classes and relationships is chosen as the AP, see Section 4.6.

3. *Mapping UML relationships to directed QPG edges.*

For each query pattern $Q_k \in \mathcal{Q}$, starting from its AP node determined in step 2, add the edges that correspond to the relationships in the query path of Q_k to E, and label them with the *aggregation label* k. If a directed edge already exists, append the *aggregation label* k to the existing set of aggregation labels. The aggregation tree labeled k is the same as the query path of Q_k at this stage of initialization. Thus, add $Q_k \to \{k\}$ to the execution plan Ω. Note that the aggregation label can be any literal and the indices of query patterns are used here for simplicity. For example, the query path of Q_3 is $Customer.requests.Order.delivers.Carrier$, and the directed edge $requests$ from $Customer$ to $Order$ and the directed edge $delivers$ from $Order$ to $Carrier$ are added to the QPG with node $Customer$ as the AP. Accordingly, these two edges are labeled 3 and the mapping $Q_3 \to \{3\}$ is added to Ω. Note that there are two edges named as *requests*, since they are in different directions in different aggregation trees.

4. *Determining the attribute functionality in G.*

For each query pattern $Q_k \in \mathcal{Q}$, add or merge SA specification pairs $[EA, \{k\}]$, $[IA, \{k\}]$, $[PA, \{k\}]$ or $[OA, \{k\}]$ to each attribute of nodes and edges based on which clause of Q_k the attribute is in. For example, every attribute in Fig. 7 has a set of SA specifications representing its functionality in the corresponding query patterns.

5. *Initializing the execution plan Ω for each query/update pattern in W.*

In step 3 above, the execution plan for query patterns is initialized. For each node $n \in N$, add $n \to p$ to Ω, where p is the union of the sets of aggregation labels in SA specifications on all non primary key attributes of n generated in 4. Note that when the primary key attribute is the only scalar attribute in an aggregation tree, the corresponding class is normalized and updates on objects of this class do not need to access this aggregation. For example, the update operations on objects of *Customer* need to access four aggregations labeled 1, 2, 3 and 4, represented as $customer \to \{1, 2, 3, 4\}$ in Ω. For each relationship $r \in R$, add $r \to q$ to Ω, where q is the union of the sets of aggregation labels associated with all edges in E that have the same name as r regardless of the edge directions. For example, the update operations on relationship *requests* need to access four aggregation trees that have two edges named *requests* in different directions, represented as $requests \to \{1, 2, 3, 4\}$ in Ω.

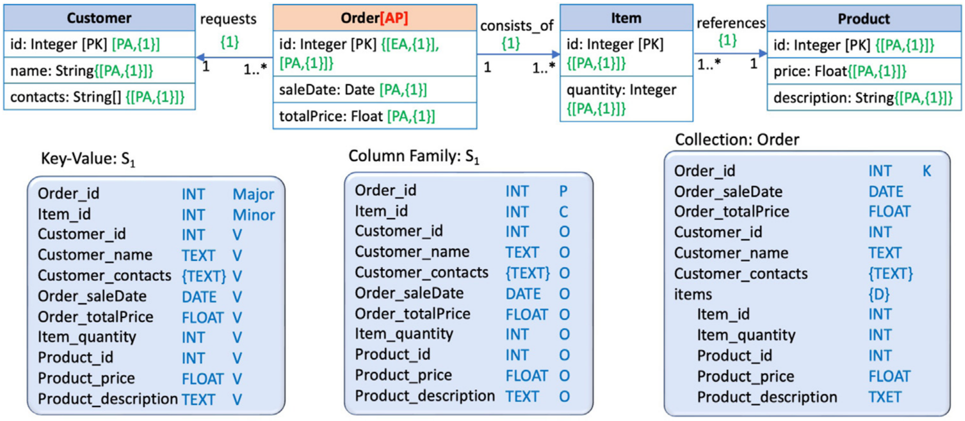


Fig. 8. KV, CF, DC schemas generated from the aggregation tree $G(1)$ in Fig. 7.

4.5. Mapping QPG schemas to NoSQL schemas

Given a QPG schema $G = (N, E, L)$, each aggregation tree $G(l)$, with $l \in L$, is mapped to a KV, CF and DC schema respectively. The following is a list of functions on the QPG schema $G = (N, E, L)$:

- $accessPoint(l)$: return the root node of the aggregation tree $G(l)$.
- $scalarAttributes(n, l)$: return a list of the scalar attributes of node n in the aggregation tree $G(l)$.
- $keyAttributes(n)$: return a list of the primary key attributes of node n.
- $children(n, l)$: return a list of the children of node n in the aggregation tree $G(l)$.
- $multiplicity(e)$: return the corresponding multiplicity of the directed edge $e = (n_i, n_j)$ on n_j's side.
- $scalarAttributes(e, l)$: return a list of the scalar attributes of edge e in the aggregation tree $G(l)$.
- $equalityAttributes(l)$: return a list of all attributes whose SA specification contains $EA : \{l\}$.
- $inequalityAttributes(l)$: return a list of all attributes whose SA specification contains $IA : \{l\}$.
- $orderingAttributes(l)$: return a list of all attributes whose SA specification contains $OA : \{l\}$.
- $projectionAttributes(l)$: return a list of all attributes whose SA specification contains $PA : \{l\}$.

4.5.1. Mapping aggregation trees to CF schemas

Given an aggregation tree $G(l)$, the algorithm $ATtoCF$ on page 19 generates the CF schema. Note that $ATtoCF$ is a recursive method that traverses the aggregation tree $G(l)$ starting from the root $accessPoint(l)$. Fig. 8 shows the CF schema S_1 generated for the aggregation tree $G(1)$ in Fig. 7, where the equality attribute $Order\_id$ is set as the partition key (P). If there are inequality attributes or ordering attributes, they will be set as the clustering keys (C) of CF S_1. $Item\_id$ is set as the clustering key (C) based on the multiplicities of the corresponding relationship, so that each row of CF S_1 is unique.

4.5.2. Mapping aggregation trees to KV schemas

Mapping an aggregation tree to a KV schema is similar to that of CF schemas. Fig. 8 shows the generated KV schema S_1 for the aggregation tree $G(1)$, where the partition key P in the CF S_1 is the major key in KV; and the clustering key C is the minor key accordingly. Thus, the cost to map an aggregation tree to KV schemas is the same as that to CF schemas.

4.5.3. Mapping aggregation trees to DC schemas

Given an aggregation tree $G(l)$, the algorithm $ATtoDC$ on page 20 traverses the tree from the root $accessPoint(l)$ and generates the DC schema. Fig. 8 shows the generated DC schema *Collection:Order* for the aggregation tree $G(1)$, where the equality attributes are set as the shard key (K) components. Node $Item$ is mapped as an array of embedded documents in S_1. If the multiplicity of a child node is one, the properties of the child node are directly attached to the parent document, instead of adding one more layer when accessing the properties, such as the nodes $Customer$ and $Product$ in Fig. 8. If a leaf node has only its primary key attributes in the aggregation tree, it is mapped to a referenced documents.

4.6. Cost modeling

NoSQL stores provide high availability and scalability by horizontal partition of the database and (to some extent) evenly distributing data units over physical servers. The increased storage usage by data denormalization can be handled by scaling out. Thus, storage constraint is not the main concern. Therefore, we use query/update cost to further optimize the specific NoSQL database schema. Note that the update cost is related to the storage usage as it depends on the number of copies.

Algorithm 1. *ATtoCF*

```
Input: A QPG schema $G$, an aggregation label $l$, a node $N$ and its parent $T$, the corresponding evolving CF schema $S_l$
Output: $S_l$ is evolved regarding $N$'s scalar attributes
for each $A \in scalarAttributes(N, l) \cup scalarAttributes((T, N), l)$ do
    // Attributes labeled as equality attributes are set as partition keys.
    if $A \in equalityAttributes(l)$ then
        Add $A$ as a $P$ component into $S_l$;
    end
    // Attributes labeled as inequality or ordering attributes are set as clustering keys.
    else if $A \in inequalityAttributes(l)$ and $A \notin S_l$ then
        Add $A$ as a $C$ component into $S_l$;
    end
    else if $A \in orderingAttributes(l)$ and $A \notin S_l$ then
        Add $A$ as a $C$ component into $S_l$ with the corresponding <+>/<-> order;
    end
    // Key attributes of node $N$ are set as clustering keys if $multiplicity((T, N))$ ='*', otherwise, as ordinary
       columns.
    else if $A \in keyAttributes(N)$ and $A \notin S_l$ then
        if ($T = \emptyset$) or ($T \neq \emptyset$ and $multiplicity((T, N))$ ='*') then
            Add $A$ as a $C$ component into $S_l$;
        end
    end
    if $A \in projectionAttributes(l)$ and $A \notin S_l$ then
        Add $A$ as a $O$ component into $S_l$;
    end
end
for each $x \in children(N, l)$ do
    $ATtoCF(G, l, x, N, S_l)$;
end
```

Given a UML class diagram $D = (C, R)$ and an application workload $W = \mathcal{U} \cup \mathcal{Q}$, we introduce two functions to measure the workload cost:

(1) $count(c_i, c_j)$, where $(c_i, c_j) \in R$, is the average number of connections that an object in class c_i has with objects in class c_j. For example, $count(Customer, Order) = 34$ and $count(Order, Customer) = 1$ in Fig. 7. Multiplicities of the corresponding relationships in R are taken as the default values of the function.

(2) $f(w_i)$, where $w_i \in W$, is the expected frequency for a query or update pattern in the application workload. We assume that query patterns have overall higher frequencies than update patterns, and $f(w_i)$ helps to detect a few outstanding classes or relationships for QPG schema optimizations.

The cost of a query or update pattern is measured using the number of read or write commands needed by the target NoSQL store to accomplish the workload. The cost of a workload $W = \mathcal{U} \cup \mathcal{Q}$ applied to the QPG schema $G = (N, E, L)$ is defined as follows:

$$cost(G) = \sum_i \sum_j (c_r(\mathcal{Q}_i, l_j) * f(\mathcal{Q}_i)) + \sum_i \sum_j (c_w(\mathcal{U}_i, l_j) * f(\mathcal{U}_i)) \tag{1}$$

where $c_r(\mathcal{Q}_i, l_j)$ is the number of reads needed by the aggregation tree $G(l_j)$ to answer query $\mathcal{Q}_i$; and $c_w(\mathcal{U}_i, l_j)$ is the number of copies the object/relationship of $\mathcal{U}_i$ has in the aggregation tree $G(l_j)$ that will be written. $c_r(\mathcal{Q}_i, l_j) = 0$ and $c_w(\mathcal{U}_i, l_j) = 0$ if the aggregation tree labeled l_j is not in the execution plan of $\mathcal{Q}_i$ and $\mathcal{U}_i$ respectively. The calculations of $c_r(\mathcal{Q}_i, l_j)$ and $c_w(\mathcal{U}_i, l_j)$ are introduced in the following sections.

Our approach starts with a set of optimal aggregation trees for all query patterns as shown in Section 4.4, which achieves the best data locality and query efficiency. Thus, every query pattern $\mathcal{Q}_k$ can be answered by one look-up on the aggregation tree $G(k)$ and $c_r(\mathcal{Q}_k, k)$ is 1 initially. On the other hand, each aggregation tree in QPG schema indicates an aggregation in the target NoSQL database, and the number of copies of objects and relationships varies based on the target NoSQL store used as shown

Algorithm 2. *ATtoDC*

```
Input: A QPG schema $G$, an aggregation label $l$, a node $N$ and its parent $T$, and $S_T$ as a hierarchical schema of $T$
Output: $S_T$ is evolved regarding $N$'s scalar attributes
Initialize $S_N$ as a schema of $N$;
for each $A \in scalarAttributes(N, l) \cup scalarAttributes((T, N), l)$ do
    Add $A$ as a property into $S_N$;
    if $A \in equalityAttributes(l)$ then
        Set $A$ as a $K$ component;
    end
end
for each $x \in children(N, l)$ do
    $ATtoDC(G, l, x, N, S_N)$;
end
if $T \neq \emptyset$ then
    if $children(N, l) = \emptyset$ and $scalarAttributes(N, l) = keyAttributes(N)$ then
        // N is a leaf node and its primary keys are the only scalar attributes in aggregation tree G(l).
        if $multiplicity((T, N)) =$'*' then
            Add $N$ as an array of referenced documents $\{* S_N\}$ into $S_T$;
        end
        else
            Add $N$ as a referenced document $* S_N$ property into $S_T$;
        end
    end
    else
        if $multiplicity((T, N)) =$'*' then
            Add $N$ as an array of embedded documents $\{D\}$ into $S_T$;
        end
        else
            Add all properties of $S_N$ into $S_T$;// Instead of adding an extra layer to the structure of $S_T$
        end
    end
end
```

in Section 4.5. A column family unnests the aggregation tree into a flattened table so that the data of an object are copied in multiple rows in a column family. A document keeps the same hierarchical structures as the aggregation tree, and one object can be copied in one or more documents in the same document collection. Therefore, for an aggregation tree $G(l_j) = (N', E', \{l_j\})$, assume there is a path from its root n_1 to a node n_i in N', represented as $(n_1 \to \ldots \to n_i)$, and the rest edges in E' are represented as $\mathcal{E}'(i) = E' - \{(n_{k-1}, n_k) | 1 < k \leq i\}$, $c_w(n_i, l_j)$ and $c_w((n_i, n_{i+1}), l_j)$ are the costs to update node n_i and edge (n_i, n_{i+1}) in aggregation tree $G(l_j)$ for a column family and a document collection defined as follows respectively:

$$\begin{cases} c_w(n_i, l_j) = \prod\limits_{1<k\leq i} count(n_k, n_{k-1}) * \prod\limits_{e \in \mathcal{E}'(i)} count(e), & (CF) \\ c_w((n_i, n_{i+1}), l_j) = \prod\limits_{1<k\leq i} count(n_k, n_{k-1}) * \prod\limits_{e \in \mathcal{E}'(i+1)} count(e), & (CF) \\ c_w(n_i, l_j) = \prod\limits_{1<k\leq i} count(n_k, n_{k-1}), & (DC) \\ c_w((n_i, n_{i+1}), l_j) = \prod\limits_{1<k\leq i} count(n_k, n_{k-1}), & (DC) \end{cases} \tag{2}$$

where $count(n_k, n_{k-1}) = 1$, if $k = 1$. Fig. 9 illustrates the calculations of c_w with an example aggregation tree rooted n_1 connecting other nodes by five edges such that $count(n_2, n_1) = a$, $count(n_1, n_2) = b$, etc. The number of copies of a node or an edge for a column family relates to the counts of all the other edges, for example $c_w(n_3, l) = c * b * f * h * n$; that is, the number of copies of node n_3 depends on all the five edges in the aggregation tree. Similarly, the number of copies of edge (n_3, n_5) depends on all the other four edges. On the other hand, the number of copies of a node or an edge for a document collection only depends on the counts of edges towards the root of the aggregation tree for example $c_w(n_5, l) = g * c$, and the repetitions of connections relate to the copies of the corresponding objects for example $c_w((n_3, n_5), l) = c_w(n_3, l) = c$. As shown in Section 4.5, the same strategies are taken for mapping an aggregation tree to a KV schema and a CF schema. Thus, the calculation of update cost for key-value stores is the same as for wide-column stores and as well as the optimization strategies.

4.7. Schema optimization

Query performance on the initial QPG schema generated in Section 4.4 is optimal as it just focuses on the query workload. Update performance can be improved by normalization at cost of query performance. If the increased query cost is amended by

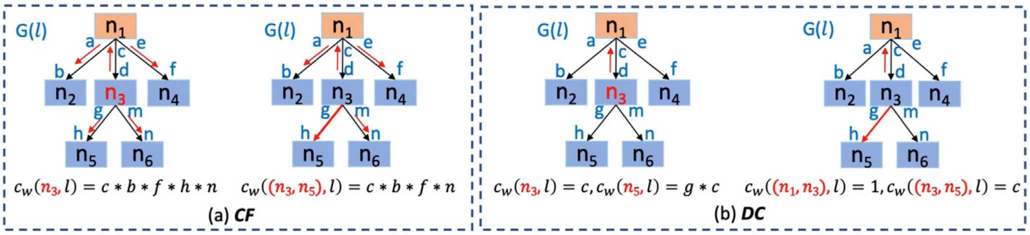
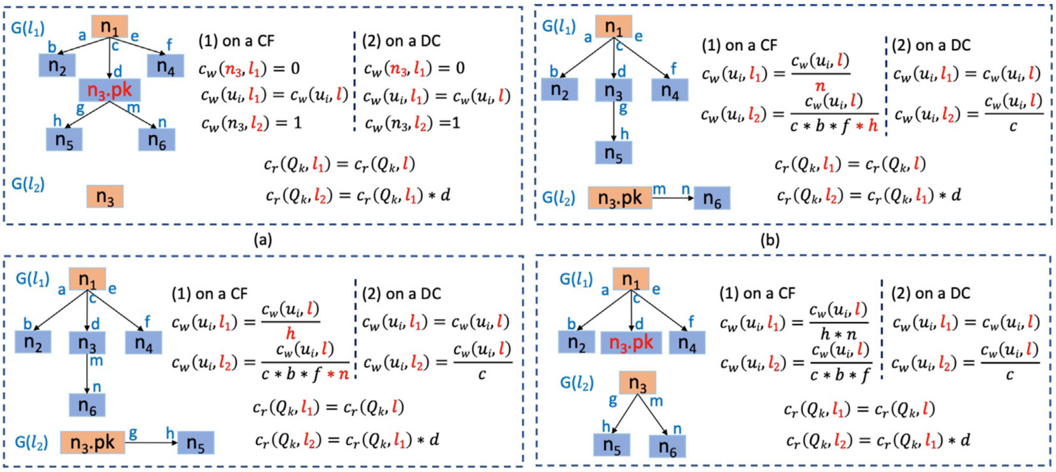

G(l)
n1
a
c
e
b
d
f
n2
n3
n4
g
m
h
n
n5
n6
$c_w(n_3, l) = c * b * f * h * n$
$c_w((n_3, n_5), l) = c * b * f * n$
(a) CF
$c_w(n_3, l) = c, c_w(n_5, l) = g * c$
$c_w((n_1, n_3), l) = 1, c_w((n_3, n_5), l) = c$
(b) DC

Fig. 9. Write cost calculation examples over a CF and a DC from an aggregation tree $G(l)$ respectively.


G(l1)
G(l2)
n1
n2
n3.pk
n3
n4
n5
n6
a
b
c
d
e
f
g
h
m
n
(1) on a CF
(2) on a DC
$c_w(n_3, l_1) = 0$
$c_w(u_i, l_1) = c_w(u_i, l)$
$c_w(n_3, l_2) = 1$
$c_w(u_i, l_1) = \frac{c_w(u_i, l)}{n}$
$c_w(u_i, l_2) = \frac{c_w(u_i, l)}{c * b * f * h}$
$c_w(u_i, l_2) = \frac{c_w(u_i, l)}{c}$
$c_w(u_i, l_1) = \frac{c_w(u_i, l)}{h}$
$c_w(u_i, l_2) = \frac{c_w(u_i, l)}{c * b * f * n}$
$c_w(u_i, l_1) = \frac{c_w(u_i, l)}{h * n}$
$c_w(u_i, l_2) = \frac{c_w(u_i, l)}{c * b * f}$
$c_r(Q_k, l_1) = c_r(Q_k, l)$
$c_r(Q_k, l_2) = c_r(Q_k, l_1) * d$

(a) (b) (c) (d)

Fig. 10. Illustrations of normalization patterns on the aggregation tree in Fig. 9.

the decreased update cost, the total cost is decreased and the schema is optimized according to Eq. (1). Note that even though we do not actively control the storage usage, data size decreased as optimizing update performance. Given an aggregation tree $G(l) = (N, E, \{l\})$ rooted $n_1 \in N$ that is split to two subtrees $G(l_1) = (N_1, E_1, \{l_1\})$ rooted n_1 and $G(l_2) = (N_2, E_2, \{l_2\})$ rooted n_2, the change of cost is calculated as follows:

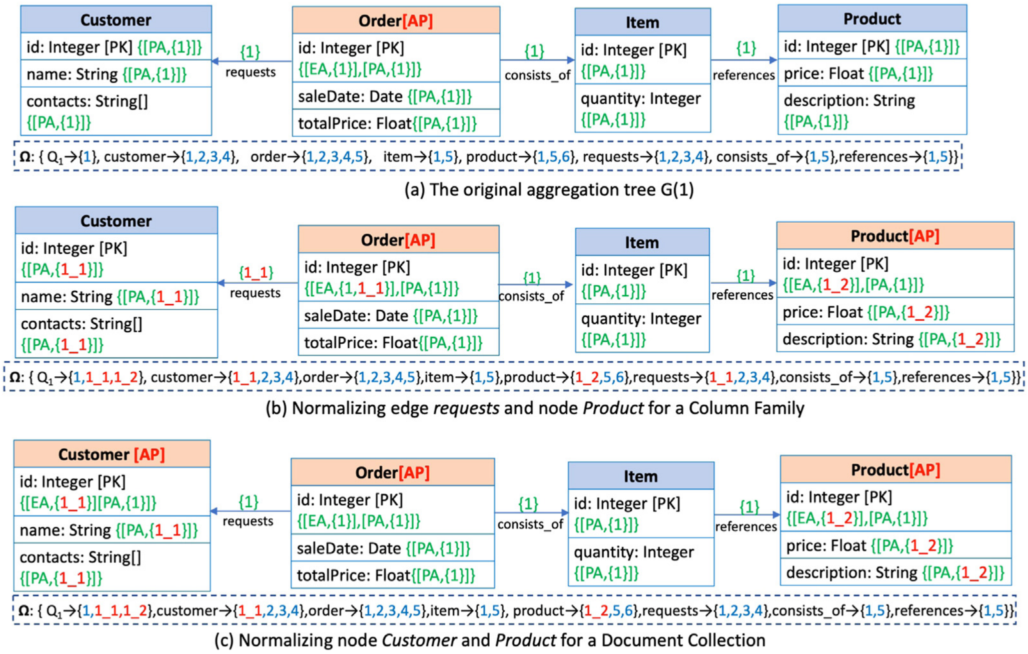
$$\begin{aligned}\triangle cost =& cost(G(l_1)) + cost(G(l_2)) - cost(G(l))\\ =& \sum_i ((c_r(Q_i, l_1) + c_r(Q_i, l_2) - c_r(Q_i, l)) * f(Q_i)) +\\ & \sum_i ((c_w(\mathcal{U}_i, l_1) + c_w(\mathcal{U}_i, l_2) - c_w(\mathcal{U}_i, l)) * f(\mathcal{U}_i))\end{aligned} \tag{3}$$

$\triangle cost$ calculates the increased query cost of all queries whose execution plan has the aggregation tree labeled l and the decreased update cost when nodes and edges in $G(l)$ are moved to smaller aggregation trees. If $\triangle cost < 0$, the total system cost is decreased.

4.7.1. Normalization

Normalizing an aggregation tree is to split it into two subtrees. Fig. 10 shows four patterns to split the aggregation tree in Fig. 9 on node n_3, and the new query/update costs for a column family and a document collection after normalization. In each pattern, the aggregation tree $G(l)$ rooted n_1 is split to two sub aggregation trees $G(l_1)$ rooted n_1 and $G(l_2)$ rooted n_3. In order to achieve the same query capability as the original aggregation tree $G(l)$, additional reads are needed on the new subtrees to join the results. As the figure shows, given a specific object of n_1 in $G(l_1)$, d different objects of n_3 are accessed. Thus, d reads are needed on the aggregation tree $G(l_2)$; that is, $c_r(Q_k, l)$ is the number of reads that query Q_k needs on the aggregation tree $G(l)$, and after splitting the tree on n_3 we can get the new query cost as $c_r(Q_k, l_1) = c_r(Q_k, l)$ and $c_r(Q_k, l_2) = d * c_r(Q_k, l_1)$. Therefore, given the aggregation tree $G(l) = (N', E', \{l\})$ with a path from the root n_1 to node n_i, represented as $(n_1 \to \ldots \to n_i)$, the query cost of Q_k on the new sub aggregation trees $G(l_1)$ rooted n_1 and $G(l_2)$ rooted n_i are defined as follows:

$$\begin{aligned}& c_r(Q_k, l_1) = c_r(Q_k, l)\\ & c_r(Q_k, l_2) = c_r(Q_k, l) * \mathop{\Pi}_{1<j\leq i} count(n_{j-1}, n_j)\end{aligned} \tag{4}$$


Customer
id: Integer [PK] {[PA,{1}]}
name: String {[PA,{1}]}
contacts: String[] {[PA,{1}]}
{1} requests
Order[AP]
id: Integer [PK] {[EA,{1}],[PA,{1}]}
saleDate: Date {[PA,{1}]}
totalPrice: Float{[PA,{1}]}
{1} consists_of
Item
id: Integer [PK] {[PA,{1}]}
quantity: Integer {[PA,{1}]}
{1} references
Product
id: Integer [PK] {[PA,{1}]}
price: Float {[PA,{1}]}
description: String {[PA,{1}]}
Ω: { Q1→{1}, customer→{1,2,3,4}, order→{1,2,3,4,5}, item→{1,5}, product→{1,5,6}, requests→{1,2,3,4}, consists_of→{1,5},references→{1,5}}
Customer
id: Integer [PK] {[PA,{1_1}]}
name: String {[PA,{1_1}]}
contacts: String[] {[PA,{1_1}]}
{1_1} requests
Order[AP]
id: Integer [PK] {[EA,{1,1_1}],[PA,{1}]}
saleDate: Date {[PA,{1}]}
totalPrice: Float{[PA,{1}]}
{1} consists_of
Item
id: Integer [PK] {[PA,{1}]}
quantity: Integer {[PA,{1}]}
{1} references
Product[AP]
id: Integer [PK] {[EA,{1_2}],[PA,{1}]}
price: Float {[PA,{1_2}]}
description: String {[PA,{1_2}]}
Ω: { Q1→{1,1_1,1_2}, customer→{1_1,2,3,4},order→{1,2,3,4,5},item→{1,5},product→{1_2,5,6},requests→{1_1,2,3,4},consists_of→{1,5},references→{1,5}}
Customer [AP]
id: Integer [PK] {[EA,{1_1}][PA,{1}]}
name: String {[PA,{1_1}]}
contacts: String[] {[PA,{1_1}]}
{1} requests
Order[AP]
id: Integer [PK] {[EA,{1}],[PA,{1}]}
saleDate: Date {[PA,{1}]}
totalPrice: Float{[PA,{1}]}
{1} consists_of
Item
id: Integer [PK] {[PA,{1}]}
quantity: Integer {[PA,{1}]}
{1} references
Product[AP]
id: Integer [PK] {[EA,{1_2}],[PA,{1}]}
price: Float {[PA,{1_2}]}
description: String {[PA,{1_2}]}
Ω: { Q1→{1,1_1,1_2},customer→{1_1,2,3,4},order→{1,2,3,4,5},item→{1,5}, product→{1_2,5,6},requests→{1,2,3,4},consists_of→{1,5},references→{1,5}}

(a) The original aggregation tree G(1)

(b) Normalizing edge *requests* and node *Product* for a Column Family

(c) Normalizing node *Customer* and *Product* for a Document Collection

Fig. 11. Examples of normalizing the aggregation tree $G(1)$ in Fig. 7.

Normalizing Nodes. Normalizing a node n from an aggregation tree $G(l)$ moves all scalar attributes of node n in $G(l)$ to a new aggregation tree. As a result, the new aggregation tree has a single node of n and only the primary key of node n is kept in $G(l)$. For example, all scalar attributes of n_3 in $G(l)$ are moved to $G(l_2)$ and only the primary key of n_3 is kept in $G(l_1)$ as shown in Fig. 10(a). Update cost of node n_3 is decreased from $c_w(n_3, l)$ to 1, represented as $c_w(n_3, l_1) = 0$ and $c_w(n_3, l_2) = 1$. Update costs of other nodes and edges remain the same as in the original aggregation tree, represented as $c_w(u_i, l_1) = c_w(u_i, l)$, where $u_i \neq n_3$.

Normalizing Edges. Normalizing an edge $e = (n_i, n_j)$ in an aggregation tree $G(l)$ moves the edge e and the corresponding subtree of n_j to a new aggregation tree. Fig. 10(b) and (c) show that one of the two out-going edges of node n_3 is split from the original aggregation tree $G(l)$. Since the execution plan of a query using $G(l)$ needs to read $G(l_1)$ first and then $G(l_2)$, the scalar attributes of node n_3 are kept in $G(l_1)$ and only the primary key of n_3 is used in $G(l_2)$.

Fig. 10(d) shows how to normalize node n_3 and its two out-going edges together. When mapping $G(l)$ to a column family, update costs of nodes and edges in $G(l_1)$ and $G(l_2)$ in Fig. 10(b), (c) and (d) are different; but when mapping $G(l)$ to a document collection, their costs are the same. Thus, when targeting document stores, every time a node and all its out-going edges, called a *normalization unit*, is normalized together; and when targeting wide-column stores, a *normalization unit* is either an edge or a node. A normalization unit determines the pattern to split an aggregation tree to two subtrees at a time. Therefore, in each iteration to optimize an aggregation tree, the $\triangle cost$ values for all candidate normalization units are calculated first according to Eq. (3) and the normalization unit with the minimal value is chosen and applied. The optimization process ends when there is no normalization unit whose $\triangle cost < 0$. Fig. 11 shows how to normalize the aggregation tree $G(1)$ in Fig. 7 for a column family and a document collection respectively. Note that updating the aggregation tree just adds or deletes the corresponding aggregation labels, and every time splitting an aggregation tree adds a new label to the subtree containing the normalization unit whereas the label for the original tree remains the same to achieve the efficiency of the algorithm. For example, an aggregation tree labeled 1_1 is added to normalize edge $requests$ in Fig. 11(b), and label 1_2 to normalize node *Product*. There is no normalization patterns for document collections that out-going edges of the same node are split to different subtrees. For example, nodes *Customer* and *Product* are split to two aggregation trees labeled 1_1 and 1_2 respectively in Fig. 11(c). Also, the execution plans for the related query and update patterns are updated accordingly.

4.7.2. Merging aggregation trees

Aggregation trees may share common nodes and edges. Therefore, they can be merged to reduce the duplication and improve the system performance if the costs of related query patterns are not increased. We define compatible aggregation trees that can be merged as follows.

Definition 7. Two aggregation trees $G(l_1)$ and $G(l_2)$ are compatible for merging into one aggregation tree $G(l')$ iff $c_r(\mathcal{Q}_i, l') \leq c_r(\mathcal{Q}_i, l_1)$, $c_r(\mathcal{Q}_i, l') \leq c_r(\mathcal{Q}_i, l_2)$, $c_w(\mathcal{U}_j, l') \leq c_w(\mathcal{U}_j, l_1)$, and $c_w(\mathcal{U}_j, l') \leq c_w(\mathcal{U}_j, l_2)$; that is, the costs of each query pattern $\mathcal{Q}_i$ and update pattern $\mathcal{U}_j$ using $G(l_1)$ and $G(l_2)$ are not increased when using $G(l')$.

Table 2
Case studies for the aggregate-oriented NoSQL stores.

Case study	Digital library	Online store	Easy cheat detection	RUBiS
#Classes	4	8	4	7
#Relationships	6	7	3	10
#Queries	9	6	5	20

Based on an aggregation tree, a column family and a document collection have different query capabilities and data duplication. Thus, the compatibility of two aggregation trees for wide-column stores and document stores are provided respectively.

Detecting Compatible Aggregation Trees for Wide-Column Stores. First, we define an ordered set $K(l)$ for an aggregation tree $G(l)$, where $K(l) = equalityAttributes(l) \cup inequalityAttributes(l) \cup orderingAttributes(l)$. Equality, inequality and ordering attributes are components of the partition and clustering keys of the corresponding CF schema as shown in Section 4.5, which determines the queries that the CF schema supports.

Two aggregation trees $G(l_1) = (N_1, E_1, \{l_1\})$ and $G(l_2) = (N_2, E_2, \{l_2\})$ are compatible when mapped to a CF schema iff (1) $G(l_1)$ and $G(l_2)$ share the same root node; (2) $K(l_1) \subseteq K(l_2)$ or $K(l_1) \supseteq K(l_2)$; (3) for all unshared edges e, where $e \in E_1$ *and* $e \notin E_2$ *or* $e \notin E_1$ *and* $e \in E_2$, $count(e) = 1$. Conditions (1) and (2) guarantee that the merged aggregation tree can answer the queries the two original aggregation trees can answer. Condition (3) guarantees that the number of copies of nodes and edges does not increase.

Detecting Compatible Aggregation Trees for Document Stores. Different from wide-column stores, adding subtrees to another aggregation tree does not affect the total duplication since the hierarchical structures are not changed. Thus, two aggregation trees $G(l_1) = (N_1, E_1, \{l_1\})$ and $G(l_2) = (N_2, E_2, \{l_2\})$ are compatible when mapped to a DC schema iff $G(l_1)$ and $G(l_2)$ share the same root node.

In addition, for more than two aggregation trees, if every two of them are compatible, they are all compatible and can be merged. Merging aggregation trees is to unify the aggregation labels of all compatible aggregation trees with the same label.

5. Implementation and evaluation

We have implemented a prototype to evaluate the mapping processes presented in this paper. It is available under a free license in an external repository.[2] As stated in the introduction, our method mainly aims to automatically generate databases for various aggregate-oriented NoSQL stores from the same conceptual schema. In this section, we evaluate our NoSQL database design approach from the following three perspectives: (1) QPG expressiveness; that is, the generic logical model QPG should be expressive enough to represent semantics of the UML class diagram and the structures of aggregate-oriented NoSQL stores; (2) schema quality; that is, as the frequencies of the query/update patterns change, the generated NoSQL schema can always achieve efficient workload performance; (3) the required computational resources of our schema generation algorithm.

5.1. QPG expressiveness

Since QPG mimics the notation of the UML class diagram, as long as relationships connecting two classes has associated multiplicities, the SQL-based query statement can use them in the *query paths* and a QPG schema that represents both the data structures and the access patterns can be properly generated. Note that special relationships, such as generalization/specification or union type, are treated as one-to-one relationships. Also, QPG captures the important query features, such as equality/inequality predicates, ordering by and project operations, which can be mapped to the corresponding constructs of the NoSQL model. Therefore, QPG can represent the query capability and data partitioning features of the target NoSQL stores.

To examine whether QPG is expressive enough to model any aggregate-oriented NoSQL database, we have modeled a set of external case studies with QPG to verify this hypothesis. Such case studies are commonly used in NoSQL research as testbeds. In particular, we have used four case studies: (1) a digital library used by Chebotko et al. [25], (2) an application in e-commerce domain used by de Lima and Mello [27], (3) a cheat detection system for multiplayer games used by Mior et al. [26], and (4) the Rice University Bidding System (RUBiS) used by Mior et al. [26] and DBSR [31] . Table 2 provides a general overview of the case studies. As typical in NoSQL stores, despite the simplicity of case studies (e.g., small number of classes), it is intended to maintain a good performance of simple queries over voluminous data in a big data context.

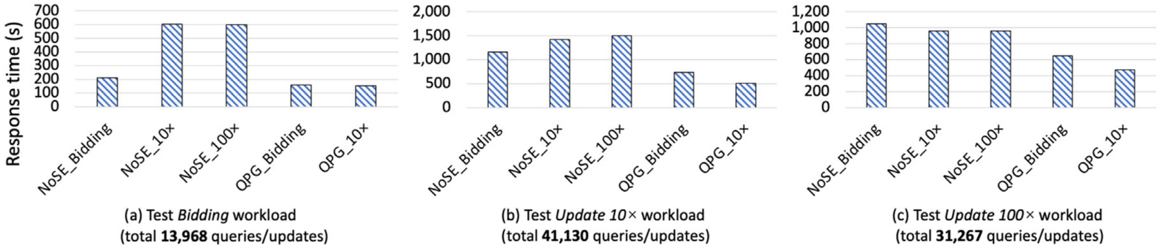
5.2. Evaluating the quality of the schemas generated

In this section, we compare the quality of the schemas generated using our method with those generated using other state-of-the-art methods. Our method is workload-driven and generates schemas for NoSQL stores based on estimated query/update frequencies and data multiplicities. In this respect, we selected NoSE [26] for wide-column stores and the method proposed by de Lima and Mello [27] for document stores for comparison.

[2] https://github.com/dollychan/QueryPathGraph-master.

Table 3
The storage usage of the five RUBiS schemas on disk.

	NoSE_Bidding	*NoSE_10*×	*NoSE_100*×	*QPG_Bidding*	*QPG_10*×
Storage usage (in GB)	7.9	7.4	7.4	7.6	5.2



(a) Test *Bidding* workload
(total **13,968** queries/updates)

(b) Test *Update 10*× workload
(total **41,130** queries/updates)

(c) Test *Update 100*× workload
(total **31,267** queries/updates)

Fig. 12. Total response time of three RUBiS workloads using five different schemas.

The experiments ran on a cluster of three Linux servers with Intel(R) Xeon (R) CPU E5-2630 v2 @2.60 GHz. The NoSQL stores are deployed on two servers, and the other is the application server that handles the join operations and possible filter operations for queries on NoSE schemas. The wide-column databases are stored in the two-node Cassandra cluster versioned 3.11.11, and the document databases in the two-node MongoDB cluster versioned 5.0.6. NoSQL databases are evenly distributed using hash partitioner and only one copy of the data is stored to reduce the effects of the replications.

To compare our method with the state-of-art works on NoSQL stores, we first generate a fake relational database and then transform it to NoSQL databases with different schemas. In this way, it is guaranteed that the NoSQL databases structured with different schemas have the same amount of information, and the performance of the same query/update request on them can reflect the quality of the schemas. Then the application server assigns values to the parameters of the query/update patterns based on the corresponding frequencies and executes the same workload over all tested NoSQL databases.

Note that secondary indexes are not considered by the state-of-art works. MongoDB supports querying on the non-key properties, thus the update requests can be executed successfully without secondary indexes. Cassandra maintains secondary indexes as separate stand-alone tables [57], which is similar to the extra CFs so-called support queries used by NoSE. Thus, for simplicity we use secondary indexes for the CF schemas generated by both NoSE and our method to access the columns on a non-key value.

5.2.1. *Case study - RUBiS on Cassandra*

Rice University Bidding System (RUBiS) [58] is a Web application benchmark originally backed by a relational database which simulates an online auction website. Same as NoSE, we considered four workloads: *Browsing* workload, *Bidding* workload, *Update 10*× workload, and *Update 100*× workload. Each workload contains a set of query and update patterns and each pattern has a relative frequency. *Browsing* workload is read-only and has only query patterns. *Bidding* workload contains 20 query patterns and 15 update patterns, and each pattern is weighted according to the relative frequencies used by NoSE. *Update 10*× and *Update 100*× workloads are based on *Bidding* workload with frequencies of all update patterns increased 10 times and 100 times respectively. We generated the CF schemas for each workloads: *QPG_Browsing*, *QPG_Bidding*, *QPG_10*× and *QPG_100*×, and compared them with the corresponding schemas generated by NoSE: *NoSE_Browsing*, *NoSE_Bidding*, *NoSE_10*× and *NoSE_100*×. For read-only *Browsing* workload, our method and NoSE generate the same schema. Also, schema *QPG_10*× and *QPG_100*× turn out to be same by our algorithms. Thus, we mainly compared the query/update performance of the five different schemas, which are *NoSE_Bidding*, *NoSE_10*×, *NoSE_100*×, *QPG_Bidding*, and *QPG_10*×.

We used the same dataset as NoSE.[3] Data are mapped to column families of the aforementioned five different schemas from the same relational database. Table 3 presents the storage usage for the five different Cassandra schemas on disk. When aggregation trees are normalized, not only the update costs are decreased but also the storage usage. Thus, the data size of schema *QPG_10*× is smaller than *QPG_Bidding*.

Fig. 12(a), (b) and (c) show the total response time of running the instances of *Bidding* workload, *Update 10*× workload and *Update 100*× workload on the five schemas respectively. We can conclude from the experiment that the schemas generated by our method have an overall better performance than the ones generated by NoSE. Fig. 13 shows the average response time for each query and update pattern using the aforementioned five schemas generated by NoSE and our method. Our method and NoSE have comparable performance for most of the query and update patterns. The differences between schemas generated by NoSE and our method mainly in the following cases.

(1) Data are normalized by our method, but not by NoSE as shown in Fig. 13 by Q_4 and Q_{16}. Our method improves the update costs for *Users* and *Items* at cost of the query performance of Q_4 and Q_{16} respectively.

[3] https://github.com/michaelmior/NoSE.

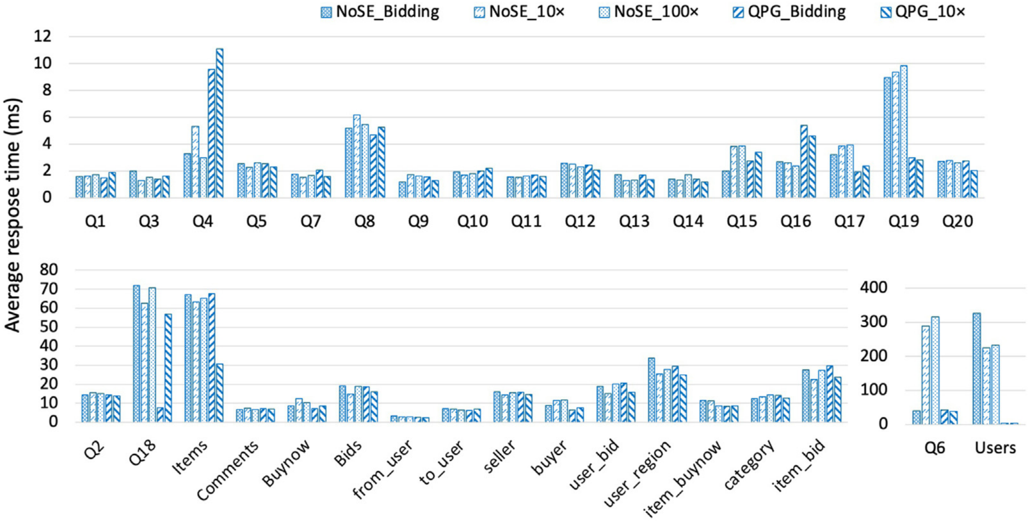


Fig. 13. Average response time of RUBiS query/update patterns using five schemas..

(2) Data are normalized by NoSE, but not by our method as shown in Fig. 13 by Q_6. The related column families for Q_6 in schema *NoSE_10*× and *NoSE_100*× normalize class *Items* compared with that in schema *NoSE_Bidding*, which largely increases the query response time. As the experiment shows, the improvement of update performance by normalizing class *Items* is not comparable to the decrease of query performance.

(3) To process the query patterns Q_{17} and Q_{19}, NoSE uses an extra $Filter$ step after reading data from column families to handle the inequality predicates. As the experiment shows, the $Filter$ process affects query performance especially when the inequality predicates is relatively selective, which means large number of unnecessary rows are read.

Also, compared with schema *QPG_Bidding*, schema *QPG_10*× normalizes *Items* in the related CF for Q_{18}, and Fig. 12(b) and (c) shows this normalization adapts to the workload change as the frequencies of updates increase. In conclusion, our method focuses on the overall system performance, rather than maximizing the performance of individual operation, and the validity of our cost model is verified by comparing the differences in schemas generated by our method and NoSE.

5.2.2. Case study - Online store on MongoDB

This case study is used to compare our method with the one proposed by de Lima and Mello [27]. The conceptual model of Online Store and 6 query patterns are shown in Section 2.1. We take the relative frequencies of query patterns {Q_1:15, Q_2:9, Q_3:4.5, Q_4:3, Q_5:1, Q_6:1} as de Lima's. The schema generated by de Lima and Mello is redundancy-free, which means the update performance is optimal. We assume the update patterns in the workload are on single class and each has a relative frequency 1, which forms the $Basic$ workload with the query patterns. Similar to the case study of RUBiS, the update frequencies are increased 10 times and 100 times to construct $Update$ 10× workload and $Update$ 100× workload respectively. It turns out that the schemas generated by our method for the three workloads are the same based on our cost model.

We used the same dataset as de Lima and Mello, which consists of 32,418 customers, 1,113,951 orders, 1,789,082 items, 900 products, 61 categories, 1,113,951 payments, 165 supplier and 16 carriers. Same as the case study of RUBiS, data are originally stored in a shared relational database and then mapped to schemas generated by de Lima's and our method. The schema generated by de Lima and Mello is 201 MB on disk compared to 534 MB for the schema generated by our method.

Fig. 15 shows the average response time for each query and update pattern using the schemas generated by de Lima's and our method. As it shows, our method performs practically the same as de Lima's for Q_3 and Q_4 as the execution plans of these queries are similar using the two schemas. Our method performs much better on Q_1, Q_2 and Q_5. For queries Q_1 and Q_2 that access a specific order, the schema generated by de Lima and Mello nests all related orders in a customer so that the target document of order is searched among documents of customers, which is time-consuming compared with the schema generated by our method that reads a document rooted the order. For query Q_5 that access all orders having a specific product, the number of documents read by our method is much less than de Lima's since the related documents are aggregated by our method. As a result, the overall update performance of our schema is worse than de Lima's. Fig. 14 shows the response time of executing the three workload instances on the two schemas. As it shows, when the frequencies of update patterns increase, the performance of the schema generated by our method becomes worse as the update commands takes up 70% and 95% of the requests in the whole workload.

We have also done the experiments to compare the DC schema generated by our method with the one in the normalized form for RUBiS, and the CF schema generated by our method with the normalized one for OnlineStore. The results show that our method generates better schemas for all read-intensive workloads. The details of the prototype system and the experiments are given in Appendix.

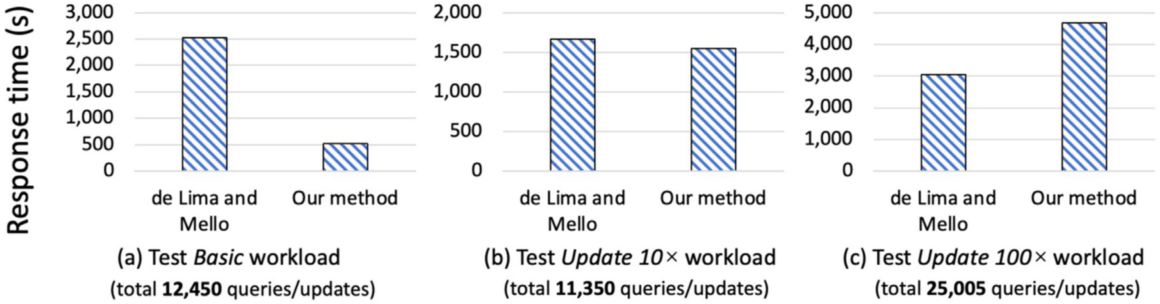


Fig. 14. Total response time of three OnlineStore workloads using two schemas.

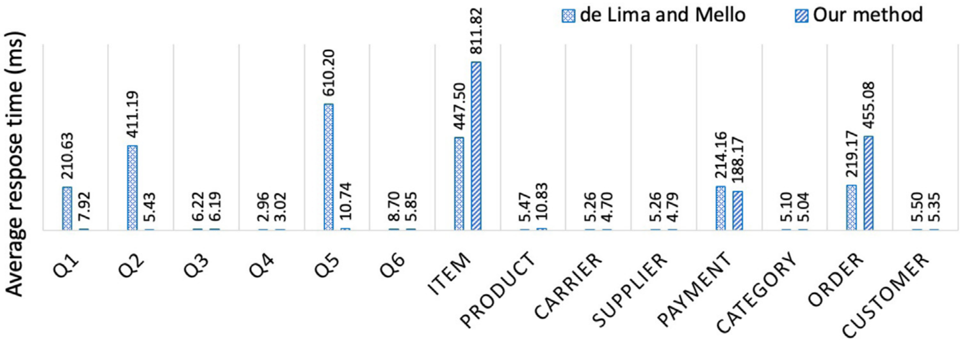


Fig. 15. Average response time of OnlineStore query/update patterns using two schemas.

5.3. Computational resources

This section discusses the computational complexity of our automatic generation process. Given a UML class diagram $D = (C, R)$ and a workload $W = \mathcal{Q} \cup \mathcal{U}$, our method consists of three steps to generate the NoSQL schemas.

(1) Generate an initial QPG schema where each query pattern is mapped to an aggregation tree. Each query pattern consist of at most $|C|$ classes and $|R|$ relationships. The time complexity of this step is $\mathcal{O}((|C| + |R|) \times |\mathcal{Q}|)$;

(2) Optimize the schemas where aggregation trees are normalized and merged. Each aggregation tree consists of at most $|C|$ nodes and $|R|$ edges as there is no cycle in the tree. When targeting key-value and wide-column stores, at most $(|R| + |C| - 1)$ iterations are needed as every node and edge of an aggregation tree may be normalized; and at most $(|C| - 1)$ iterations are needed when targeting document stores as all non-root nodes are checked. Thus, the complexity of the normalization process is $\mathcal{O}((|R| + |C| - 1) \times |\mathcal{Q}|)$ or $\mathcal{O}((|C| - 1) \times |\mathcal{Q}|)$. After normalization, each original aggregation tree may be partitioned to at most $|R| + |C|$ aggregation trees when targeting key-value and wide-column stores, and at most $|C|$ aggregation trees when targeting document stores. Every two of the aggregation trees are compared to check if they are compatible and compatible aggregation trees are merged. Thus, the complexity of the merging process is $\mathcal{O}(((|R| + |C|) \times |\mathcal{Q}|)^2)$ or $\mathcal{O}(((|C|) \times |\mathcal{Q}|)^2)$. Based on the merging rules, all aggregation trees for key-value and wide-column stores may not be merged due to incompatible query capabilities, and at most $|C|$ aggregation trees would be generated for document stores;

(3) Map each aggregation tree to a target NoSQL schema. The complexity of this step is $\mathcal{O}((|R| + |C|) \times |\mathcal{Q}|)$ or $\mathcal{O}(|C|)$.

In normal scenarios, both the values $|C|$, $|R|$ and $|Q|$ are expected to be low, and the complexity of the above steps is polynomial. The times are about a few seconds to generate database schemas in the aforementioned case studies.

6. Conclusion

This paper has presented a novel workload-driven method for designing the schemas for different aggregate-oriented NoSQL databases from the same conceptual schema. A generic logical NoSQL model called Query Path Graph (QPG) is used to represent the data structure as the UML class diagram. Mapping rules are defined to transform the query patterns represented as SQL-based queries into aggregation trees in the QPG model. Also, mappings between the QPG model and three aggregate-oriented NoSQL data models (i.e., KV, CF and DC) are defined, which reflects the features of the data model, query capability, and the strategy of data partitioning and scalability of the NoSQL stores. In addition, a cost model is used to improve the quality of the QPG schema as it makes a trade-off between the query and update costs, which in the end optimizes the generated NoSQL schemas. We have implemented a prototype system for our method. The quality of the generated schemas and the validity of our cost model are evaluated and verified by comparing our method with other state-of-art workload-driven NoSQL schema generation methods.

Instead of the class-level update patterns, we would like to improve the cost model to measure the update costs on the attribute-level so that fine-granular schema optimization strategies can be adapted. Also, we would like to introduce secondary indexes to our cost model to improve the performance for both query and update requests.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for their critical reading of the article and their valuable feedback, which has substantially helped to improve the quality and accuracy of this article. This work was partly supported by Guangzhou Key Laboratory of Big Data and Intelligent Education (No. 201905010009) and the National Natural Science Foundation of China (No. 61672389).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.datak.2022.102089.

References

[1] D.J. Abadi, Data management in the cloud: Limitations and opportunities, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 32 (1) (2009) 3–12.
[2] A. McAfee, E. Brynjolfsson, T.H. Davenport, D. Patil, D. Barton, Big data: the management revolution, Harvard business review 90 (10) (2012) 60–68.
[3] A. Davoudian, M. Liu, Big data systems: A software engineering perspective, ACM Computing Surveys (CSUR) 53 (5) (2020) 110.
[4] M. Stonebraker, SQL databases v. NoSQL databases, Communications of the ACM 53 (4) (2010) 10–11.
[5] R. Hecht, S. Jablonski, NoSQL evaluation: A use case oriented survey, in: 2011 International Conference on Cloud and Service Computing, IEEE, 2011, pp. 336–341.
[6] R. Cattell, Scalable SQL and NoSQL data stores, ACM SIGMOD Record 39 (4) (2011) 12–27.
[7] P.J. Sadalage, M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Pearson Education, 2013.
[8] A. Davoudian, L. Chen, M. Liu, A survey on NoSQL stores, ACM Computing Surveys (CSUR) 51 (2) (2018) 40.
[9] J. Gray, P. Helland, P. O'Neil, D. Shasha, The dangers of replication and a solution, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 1996, pp. 173–182.
[10] P. Helland, Life beyond distributed transactions: An apostate's opinion, Queue (ISSN: 1542-7730) 14 (5) (2016) 69–98.
[11] W. Vogels, Eventually consistent, Communications of the ACM 52 (1) (2009) 40–44.
[12] A. Fekete, D. Guptab, V. Luchangcob, N. Lynchb, Eventually-serializable data services, in: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, 1996, pp. 300–309.
[13] E. Evans, Domain-Driven Design: Tacking Complexity in the Heart of Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.
[14] T. Vajk, P. Fehér, K. Fekete, H. Charaf, Denormalizing data into schema-free databases, in: 2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom), IEEE, 2013, pp. 747–752.
[15] V.C. Storey, I.-Y. Song, Big data technologies and management: What conceptual modeling can do, Data & Knowledge Engineering 108 (2017) 50–67.
[16] N. Bruno, S. Chaudhuri, Constrained physical design tuning, Proceedings of the VLDB Endowment 1 (1) (2008) 4–15.
[17] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, M. Syamala, Database tuning advisor for microsoft SQL server 2005, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005, pp. 930–932.
[18] S. Agrawal, E. Chu, V. Narasayya, Automatic physical design tuning: Workload as a sequence, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 2006, pp. 683–694.
[19] D.C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, S. Fadden, DB2 design advisor: Integrated automatic physical database design, in: Proceedings of the 13th International Conference on Very Large Data Bases (VLDB), 2004, pp. 1087–1097.
[20] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR) 15 (4) (1983) 287–317.
[21] H. Vera-Olivera, R. Guo, R.C. Huacarpuma, A.P.B. Da Silva, A.M. Mariano, M. Holanda, Data modeling and NoSQL databases-a systematic mapping review, ACM Computing Surveys (CSUR) 54 (6) (2021) 1–26.
[22] G.A. Schreiner, D. Duarte, R. dos Santos Mello, SQLtoKeyNoSQL: a layer for relational to key-based NoSQL database mapping, in: Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services, 2015, pp. 1–9.
[23] C.J.F. Candel, D.S. Ruiz, J.J. García-Molina, A unified metamodel for NoSQL and relational databases, Information Systems 104 (2022) 101898.
[24] X. Li, Z. Ma, H. Chen, QODM: A query-oriented data modeling approach for NoSQL databases, in: Workshop on Advanced Research and Technology in Industry Applications (WARTIA), IEEE, 2014, pp. 338–345.
[25] A. Chebotko, A. Kashlev, S. Lu, A big data modeling methodology for Apache Cassandra, in: International Congress on Big Data, IEEE, 2015, pp. 238–245.
[26] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, NoSE: Schema design for NoSQL applications, IEEE Transactions on Knowledge and Data Engineering 29 (10) (2017) 2275–2289.
[27] C. de Lima, R.S. Mello, On proposing and evaluating a NoSQL document database logical approach, International Journal of Web Information Systems 12 (4) (2016) 398–417.
[28] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the NoSQL world, Computer Standards & Interfaces 67 (2020) 103149.
[29] C. Li, Transforming relational database into HBase: A case study, in: International Conference on Software Engineering and Service Sciences, IEEE, 2010, pp. 683–687.
[30] T. Jia, X. Zhao, Z. Wang, D. Gong, G. Ding, Model transformation and data migration from relational database to MongoDB, in: International Congress on Big Data, IEEE, 2016, pp. 60–67.

[31] V. Reniers, D. Van Landuyt, A. Rafique, W. Joosen, A workload-driven document database schema recommender (DBSR), in: International Conference on Conceptual Modeling, Springer, 2020, pp. 471–484.

[32] F. Abdelhédi, A.A. Brahim, F. Atigui, G. Zurfluh, UMLtoNoSQL: Automatic transformation of conceptual schema to NoSQL databases, in: 14th International Conference on Computer Systems and Applications (AICCSA), IEEE, 2017, pp. 272–279.

[33] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla, P. Sánchez, Mortadelo: Automatic generation of NoSQL stores from platform-independent data models, Future Generation Computer Systems 105 (2020) 455–474.

[34] J. Lu, I. Holubová, Multi-model databases: A new journey to handle the variety of data, ACM Computing Surveys (CSUR) 52 (3) (2019) 1–38.

[35] U. Störl, M. Klettke, S. Scherzinger, NoSQL schema evolution and data migration: State-of-the-art and opportunities, in: Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), 2020, pp. 655–658.

[36] J. Mali, F. Atigui, A. Azough, N. Travers, ModelDrivenGuide: An approach for implementing NoSQL schemas, in: International Conference on Database and Expert Systems Applications, Springer, 2020, pp. 141–151.

[37] Oracle Inc, Oracle NoSQL database ([Online]). URL https://www.oracle.com/database/technologies/related/nosql.html.

[38] A. Inc, What is Cassandra? ([Online]). URL http://cassandra.apache.org/.

[39] MongoDB Inc, The database for modern applications ([Online]). URL https://mongodb.org/.

[40] Oracle Inc, Interface TableAPI ([Online]). URL https://docs.oracle.com/database/nosql-12.1.3.5/javadoc/oracle/kv/table/TableAPI.html.

[41] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2) (2008) 1–26.

[42] Apache Inc, Welcome to Apache HBase ([Online]). URL https://hbase.apache.org/.

[43] MongoDB Inc, BSON ([Online]). URL http://bsonspec.org/.

[44] JSON Organization, JSON Schema ([Online]). URL https://json-schema.org/.

[45] M. Lawley, R.W. Topor, A query language for EER schemas, in: Australasian Database Conference, 1994, pp. 292–304.

[46] G. Daniel, G. Sunyé, J. Cabot, UMLtoGraphDB: mapping conceptual schemas to graph databases, in: International Conference on Conceptual Modeling, Springer, 2016, pp. 430–444.

[47] R. Schroeder, D. Duarte, R.S. Mello, A workload-aware approach for optimizing the XML schema design trade-off, in: 13th International Conference on Information Integration and Web-Based Applications and Services, 2011, pp. 12–19.

[48] J. Akoka, I. Comyn-Wattiau, Roundtrip engineering of nosql databases, Enterprise Modelling and Information Systems Architectures (EMISAJ) 13 (2018) 281–292.

[49] A.A. Brahim, R.T. Ferhat, G. Zurfluh, Model driven extraction of NoSQL databases schema: Case of MongoDB, in: KDIR, 2019, pp. 145–154.

[50] F. Abdelhedi, A.A. Brahim, R.T. Ferhat, G. Zurfluh, Reverse engineering approach for NoSQL databases, in: International Conference on Big Data Analytics and Knowledge Discovery, Springer, 2020, pp. 60–69.

[51] D.S. Ruiz, S.F. Morales, J.G. Molina, Inferring versioned schemas from NoSQL databases and its applications, in: International Conference on Conceptual Modeling, Springer, 2015, pp. 467–480.

[52] A.H. Chillón, D.S. Ruiz, J.G. Molina, S.F. Morales, A model-driven approach to generate schemas for object-document mappers, IEEE Access 7 (2019) 59126–59142.

[53] F. Abdelhedi, A.A. Brahim, F. Atigui, G. Zurfluh, MDA-based approach for NoSQL databases modelling, in: International Conference on Big Data Analytics and Knowledge Discovery, Springer, 2017, pp. 88–102.

[54] W. Winston, Introduction to Mathematical Programming: Applications and Algorithms, Duxbury, (2002), Duxbury, 2002.

[55] T. Halpin, T. Morgan, Information Modeling and Relational Databases, Morgan Kaufmann, 2010.

[56] O.G. Tsatalos, M.H. Solomon, Y.E. Ioannidis, The GMAP: A versatile tool for physical data independence, Int. J. Very Large Data Bases (VLDB) 5 (2) (1996) 101–118.

[57] M.A. Qader, S. Cheng, V. Hristidis, A comparative study of secondary indexing techniques in LSM-based NoSQL databases, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 551–566.

[58] E. Cecchet, J. Marguerite, W. Zwaenepoel, Performance and scalability of EJB applications, ACM SIGPLAN Notices 37 (11) (2002) 246–261.

Liu Chen is a Ph.D Candidate in the Cotutelle program between the School of Computer Science at Wuhan University and the School of Computer Science at Carleton University. She received her bachelor's. degree in software engineering from Wuhan University in 2011. Her research interests include NoSQL data modeling and stream data processing.

Ali Davoudian is a Cloud Infrastructure Engineer at Canada Revenue Agency (CRA). He received his bachelor's and master's degrees in software engineering from Isfahan University and Amirkabir University of Technology in 2004 and 2007 respectively, and Ph.D. degree from the School of Computer Science at Carleton University in 2021. His research interests include NoSQL data modeling and big data analytics.

Mengchi Liu is a professor in the School of Computer Science at South China Normal University. He received his Bachelor's and Master's degrees in Computer Software from Wuhan University in China in 1983 and 1986 respectively, and Master's and Ph.D. degrees in Computer Science from the University of Calgary in 1990 and 1992 respectively. His research interests include data models, database systems, and big data management and analytics. He has published over 160 papers in various international journals and conferences, and served as program committee and organizing committee member for a number of international conferences and a reviewer for 15 distinguished international journals.