The Journal of Supercomputing (2024) 80:4000–4048
https://doi.org/10.1007/s11227-023-05613-5

# Schema generation for document stores using workload-driven approach

**Neha Bansal[1] · Shelly Sachdeva[1] · Lalit K. Awasthi[2]**

Accepted: 17 August 2023 / Published online: 8 September 2023


## Abstract

Although there are numerous data modeling tools for relational databases, data modeling for NoSQL databases has seen another perspective. These databases (a) do not define any explicit schema, (b) store data in a denormalized manner, and (c) give many structure alternatives. The decision to structure the data always relies on rules of thumb, which do not guarantee an optimal structural solution. Based on this motivation, this paper offers a workload-driven model for the logical schema design of a NoSQL document database. It consists of Model input, Intermediate transformation, and Final schema generation. The proposed model takes the conceptual schema (EER model) and application workload (queries and anticipated data volume) as input and describes a procedure to convert it into a logical model for NoSQL document stores. The conversion process initially converts the application queries into query graphs. The query graphs, along with the anticipated data volume, are used to generate the query labels. The resulting query labels are assigned on the schema graph designed from the EER model. The schema graph and labels are used to transform the EER model into the appropriate logical schema model based on the actions defined for each label. We evaluate the model using a case study in the eCommerce application domain. The experimental evaluation shows the proposed model outperforms the existing conventional, optimized, and query path graphs models in multiple aspects, including query performance, storage space efficiency, aggregate pipeline efficiency, read–write latency, collection-wise performance, scalability, throughput and latency. By effectively addressing the challenges associated with managing the variety and volume of big data through a well-designed schema, our proposed model significantly reduces the time, cost, and effort required for schema development and repair.

**Keywords** NoSQL database · Data modelling · Workload-driven approach · MongoDB · Schema design

Extended author information available on the last page of the article

## 1 Introduction

With the rise of big data, the requirement of applications to change their schema is more frequent and crucial. This demand has given rise to the emergence of NoSQL databases, a new category designed to overcome the limitations of traditional relational databases in handling big data and real-time applications characterized by high-speed data generation (volume) and diverse data formats (variety). NoSQL is an umbrella term used for numerous non-relational database types. Four popular categories of NoSQL are named document-based, column-based, key-value-based, and graph-based [1, 2]. These four categories share similar logical structures: A key followed by a value; however, they are distinct in data modeling, data architecture, querying languages, and API's. Typically, the performance of these four categories depends on the selection of use cases.

Unstructured data collected from sources like sensors, social media, and natural language processing (NLP) holds valuable insights [3]. To extract valuable insights from unstructured data, new data storage solutions like Hadoop and NoSQL databases have emerged [4]. These technologies are extensively applied in domains, such as the Internet of Things, Facebook, Google, and Netflix [5, 6]. The increasing adoption of NoSQL databases in handling big data is driven by their ability to manage massive volumes of data without a predefined schema. NoSQL databases, excel in handling unstructured and semi-structured data aligning with the variety criterion of big data. Horizontal scalability using sharding and replication [7] is another key aspect addressed by NoSQL databases, allowing data distribution across multiple nodes to accommodate large volumes. The schema flexibility and horizontal scalability properties ensures efficient storage and processing without compromising performance. The proposed work is aligned with the variety and volume criteria of big data.

Although NoSQL database flexibility enables rapid initial development so that the application does not need to define a specific structure in advance [6, 8], the decision should be made early because (a) The application's overall performance depends on the schema choice selection. The wrong choices can impact several aspects of application quality, like data redundancy, navigation cost, data access cost, and maintainability. (b) It is challenging to fix a poorly designed data model after the development of an application. (c) For a poorly designed data schema, it is possible that some queries require excessive execution time or cannot be executed at all. Therefore, it is preferable to spend some time in advance designing a data model that is scalable, extensible, and maintainable throughout the application's lifetime.

The flexibility of NoSQL databases empowers developers and organizations to store and manipulate data according to their specific requirements. As a result, there can be numerous schema alternatives to model the same information [9]. Analyzing and comparing these schema alternatives can be complex and time-consuming using manual methods [8, 10]. Thus, there is a need for an automated tool or model that can evaluate various factors and can recommend optimal schema solutions from the available alternatives. Two existing approaches give automation to perform this task: Workload-Agnostic and Workload-Driven.

The Workload-Agnostic approach [11–13] focuses on creating the database schema without considering any specific workload or usage patterns. The goal is to develop a schema that can handle a variety of queries and workloads. The objective is to offer flexibility and adaptability to handle various queries and data. However, this approach may not optimize performance for particular query patterns or workloads because it does not consider the specific query characteristics. On the other hand, in a Workload-Driven approach [10, 14–17], the database schema is created for the specific workload or usage patterns. The schema design is influenced by the types of queries expected to be executed frequently, the data access patterns, and the workload's performance requirements. The goal is to optimize the schema design to improve query performance, reduce latency, and improve the overall system's efficiency. In our study, we have chosen a workload-driven approach to design an automated model that considers the workload queries and anticipated data volume to provide an optimal schema solution. We intend to design a schema that best meets the performance requirements and efficiency goals by analyzing the query characteristics of the workload.

This paper has proposed an automated model to transform the conceptual model into an optimal logical schema design with the aid of labels. It consists of three parts: Model input, Intermediate transformation, and Final schema generation. Model input consists of the EER model as well as the application workload. The application queries and the estimated data volume comprise the application workload. The intermediate transformation includes the generation of query graphs and the generation of query labels. The application queries are first transformed into query graphs, and then the generated query graphs are transformed into query labels using data volume. The generation of query labels involves three steps Label Categorization, Action Association, and Prioritization. The final schema generation consists of two parts: a) the generation of a Schema Graph and Label assignment, b) transformation into Logical Schema. The EER model is first converted into a graph model named schema graph. Then the derived query labels are assigned on the edges of the schema graph. Finally, the schema graph and labels are used to transform the EER model into an optimized logical schema based on the actions defined for each label. The working of the proposed model is evaluated through a case study in the eCommerce domain. We have picked MongoDB to work on because it is the most popular store among all document stores [18]. In addition, it is used in various applications, including eCommerce, mobile applications, and many more.

In this paper, we have made the following significant contributions:

(a) The paper uses application workload to generate NoSQL document logical schemas from the conceptual model. The workload information is provided by the designer in terms of estimated total data volume and queries.
(b) The proposed model uses query graphs, query labels, and schema graphs to transform conceptual inputs into logical schemas.
(c) Query graphs are generated from workload queries and are used to analyze query characteristics. Query labels are used to showcase the investigated query characteristics.

(d) The derived query labels and the schema graph are used to design the logical schema for NoSQL document stores.
(e) To evaluate the proposed model, experiments are conducted through a case study in the eCommerce domain to showcase the performance of the proposed model.
(f) The results show the proposed model reduces query response time and accelerates the data retrieval time of workload queries.

The remainder of the paper is arranged in the following sections. Section 2 gives the related work; Sect. 3 presents the detailed work of the proposed model. Section 4 presents the experimental evaluation, and Sect. 5 concludes the paper.

## 2 Related work and motivation

In the realm of Big Data applications, the large volume, variety, and velocity of data often surpass the capabilities of traditional relational databases [19]. NoSQL databases, such as MongoDB, Cassandra, HBase, and Neo4j, have emerged as vital technologies to overcome these challenges. These databases offer flexible data models, horizontal scalability, and high-performance data processing, making them well-suited for managing massive amounts of data in distributed environments. NoSQL databases are particularly well-suited for managing heterogeneous data due to their flexible models, large volumes' scalability, and high data retrieval performance [5, 6]. Distributed databases support supercomputing by providing the necessary infrastructure and capabilities for large-scale data processing and high-performance computing workloads [20–23].

Many tools are available in the market for data modeling of traditional databases (such as relational) [24, 25]. Still, these tools cannot be applied directly to the NoSQL database due to data modeling differences (normalized versus denormalized format, respectively). Authors [26, 27] comprehensively analyze the design requirements of NoSQL databases. Uta et al. [28] have presented various case studies on top-down, bottom-up, and reverse engineering approaches for schema management in NoSQL databases. According to Paola Gomez et al. [29], the performance of a NoSQL system is determined by appropriate schema design selection among all the design options. Similarly, Mior [30] stated that the performance of a NoSQL database depends on the choice of an appropriate schema design. They proposed a manual cost-based model based on workload queries for the physical optimization of column-based data stores. However, choosing the best suitable schema among all the possible schema alternatives (schema optimization) is difficult to perform manually. From this initial study, we find the following research gaps:

(a) Unlike a relational database, the NoSQL database allows various data structure alternatives, which remains an ongoing research problem. Numerous researchers are working in this field [9, 16, 17, 31, 32].
(b) NoSQL databases inherent flexibility and schema-less nature give rise to multiple schema design alternatives. For example, consider a scenario, if there are two entities representing student (S) and their faculty (F) related by a one-to-many

relationship ($r_1$). Relationship ($r_1$) can be materialized by nesting or referencing information from the related entities. Hence there are multiple ways of schema design (S1 to S8) to store this information in document stores, as shown in Fig. 1 (adapted from [9]). The choice among these different schema designs depends on many factors, like data retrieval costs, query access patterns, and user needs. Manual schema design, typically guided by trial-and-error or ad-hoc methods, can be time-consuming and lacks a guarantee of optimal design among the various alternatives. A recent study [31] has found that only 9% of the database experts identified the optimal design among these possibilities. This evidence shows that the current manual way of database design does not yield the expected results, even for minimal scenarios taken as an example. Consequently, automation becomes crucial in streamlining the complex process, reducing time requirements, and selecting the most optimal schema design from the available options.

(c) Numerous researchers have adopted different methodologies to convert conceptual to logical schema design. We have studied the existing working models and made the comparison based on common characteristics named conceptual schema, additional inputs, conversion methodology, target model, and automation, as shown in Table 1. We have categorized the existing work into the Workload-Agnostic (WA) and Workload-Driven (WD) approaches. WA does not consider the application workload means that the schema is designed without considering the specific queries or operations that the application can perform on the database. In contrast to WA, WD considers the application workload for NoSQL schema design. These methodologies consider the specific workload requirements, such as the types of queries, patterns, or operations the application is expected to perform on the database. By considering the workload, the schema can be optimized to support the application's specific needs better and improve performance.

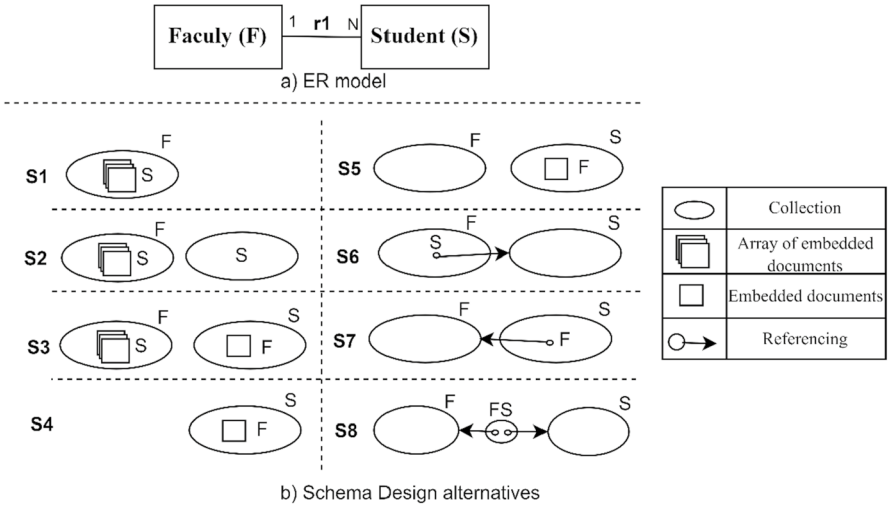


**Fig. 1** Schema design alternatives (S1-S8) in Document stores for ER model

**Table 1** Comparison of related work for NoSQL data modeling

| WA/WD | Reference | Conceptual schema | Additional inputs | Conversion methodology | Target model | Automation |
|---|---|---|---|---|---|---|
| Workload agnostic | Li [11] | ✖ | Relationships | Heuristics | C | ✖ |
| | Cerenak et al. [12] | ✖ | Relationships | Heuristics | D | ✖ |
| | Imam et al. [33, 34] | ✖ | Entities, CRUD operations | Mechanical | D | ✓ |
| | Imam et al. [13] | ✖ | Relationships | Heuristics | D | ✖ |
| Workload driven | Chebotko et al. [35] | ER | ERQL queries | Heuristics | C | ✓ |
| | Jia et al. [36] | ER | Query log | Schema graph, DAGs, tags | D | ✓ |
| | Mior et al. [10] | ER | SQL queries | Schema graph | C | ✓ |
| | Lima et al. [14, 37] | EER | Estimated data volume | Heuristics | D | ✓ |
| | Reniers et al. [38] | ER | SQL queries | Heuristics | D | ✖ |
| | Ali et al. [17, 39] | ER | SQL, query pattern | Query graph | D, C, K | ✓ |
| | Hewasinghage et al. [40, 41] | ER | Storage requirements | Canonical representation | D | ✓ |
| | Paola et al. [9] | UML | Workload queries | Software product lines strategies based on feature models | D | ✓ |
| | Proposed work | EER | Relationships, workload queries, estimated data volume | Query graph, Schema graph, Query labels | D | ✓ |

*WA* Workload-agnostic; *WD* Workload-driven; *K* Key-value, *C* Column-store, *D* Document-based

### 2.1 Workload-agnostic (WA) approach

Li [11] gives the heuristics for converting the relational schema into an HBase (NoSQL column Store) schema. Similarly, Authors [12] have designed a heuristics-based method for converting RDB to document stores using relationship types. Imam et al. [33, 34] propose a mechanical schema suggestion model for a document database. Imam et al. [13] have given manual heuristics-based guidelines to translate ER model to Document stores using relationship type and cardinality. The drawback of the existing works is that they are workload-agnostic (WA), which means they do not consider application workload. Therefore, do not guarantee to give the best optimal schema design solution and can hamper the application's performance.

### 2.2 Workload-driven (WD) approach

Chebotko [35] offers the first workload-driven (WD) design method for mapping the ER model to Cassandra (NoSQL column Store). The mapping was done based on the application workflow by taking the ERQL queries. The proposed technique improves the performance of reading operations while decreasing the performance of write operations. Tianyu Jia et al. [36] have used graphs and DAGs during schema migration from relational schema to MongoDB. The graph is generated with the help of some tags, and the relational logs are used to define the tags on the ER model. The authors have used a threshold to calculate the tags, which seems bogus due to a lack of threshold information. Mior et al. [10] have proposed a tool for schema design recommendations for column stores (C). They use ER model along with the workload queries. The query frequencies and volume of data in each candidate plan are analyzed to suggest the best solution. However, the work applies to column stores only. Authors [14, 37] provide a logical mapping from the conceptual model (EER (Extended Entity-Relationship)) using initial workload information (in terms of the estimated number of data instances and primary query operations). They developed several rules based on workload data to map the entities and relationships from EER to MongoDB. But the authors have considered workload in the form of data volume only, which is insufficient to design an optimal schema. Vincent Reniers et al. [38] use the MongoDB schema to generate workload queries and ER model. The authors also considered various dimensions while schema generation, but the model and methodologies are not given too clearly and are not automated.

Similarly, Ali et al. [17, 39] have designed the schema recommendation model based on query patterns. The authors translate the workload queries into query path graphs. The query path graphs are then translated into logical schema using various rules designed by the authors. They have performed the embedding in the case of document stores and have not considered referencing during denormalization. Authors [40, 41] have used canonical representation to suggest the denormalized model using application queries. The proposed model only applies to document stores and is very complicated to be adopted by novice users due to estimated storage space as an input requirement.

Similarly, Paola et al. [9] studied various data structuring alternatives using software product line strategies and feature models. They developed a set of structural metrics to analyze the characteristics of these alternatives. Their work aimed to propose a model that enables the automatic generation of multiple suitable data structure alternatives based on an initial UML model. The challenge of the work is accurately analyzing data structuring alternatives and generating a comprehensive set of suitable options while considering various factors like performance, scalability, and system maintainability.

Based on the current works, as stated in Table 1, both WA and WD approaches have advantages and considerations. WA approaches offer flexibility and adaptability to varying workloads, but they do not guarantee to optimize the schema among various alternatives. Based on the WD approach, the existing work can provide more targeted optimal schema solutions but require a good understanding of the application workload. The work done so far for the WD approach considers workload queries [9, 10, 17, 36, 40] or estimated data volume [14, 37] as input. However, to generate an efficient NoSQL schema using a Workload-Driven (WD) approach, it is also necessary to consider the application workload in terms of workload queries and estimated data volumes. To fill the gap in the literature, we developed a schema generation model based on a workload-driven approach that considers application workload in the forms of workload queries and data volume to generate the schema, especially for document stores.

## 3 Schema generation for document stores using workload-driven approach

This section details the proposed schema generation model for document stores using the workload-driven approach. As shown in Fig. 2, the proposed model consists of three parts: Model input, Intermediate transformation, and Final schema generation. The proposed model begins with a conceptual model and application workload as input and produces the logical schema as output using an intermediate transformation. The graphical flow diagram is shown in Fig. 3, which shows how three parts of the model (shown in Fig. 2) work together. It is intended to be used for document stores during the early stages of application development.

### 3.1 Model input

The proposed model takes a conceptual model in the form of an EER model and application workload in the form of workload queries and expected data volume as input. The details about model input are mentioned in this section.

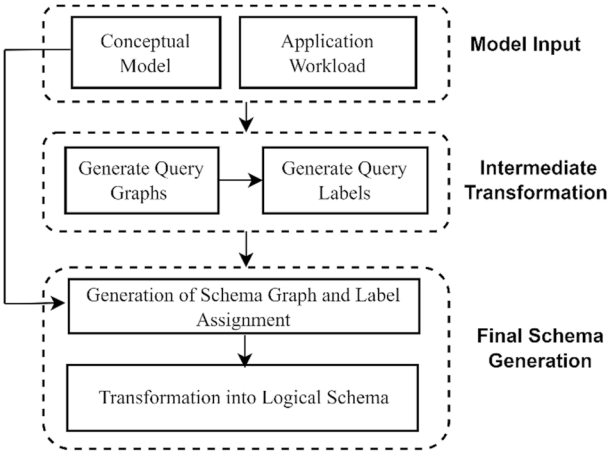


**Fig. 2** Workload-driven approach for Document Stores

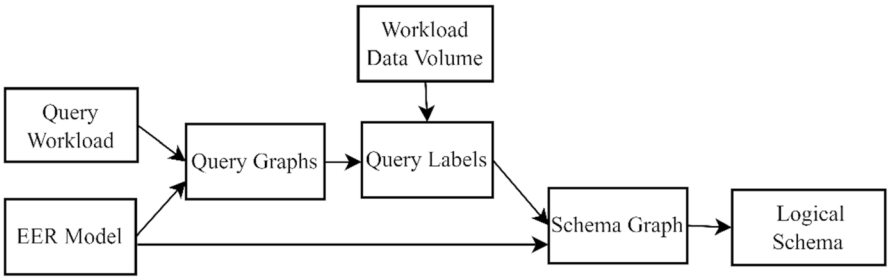


**Fig. 3** Graphical flow model

### 3.1.1 Conceptual model

The conceptual model comprehensively captures the application requirements and workflow and represents the information in a high-level abstraction model in entities, relationships, and constraints. The conceptual modeling employs numerous techniques, including ER (Entity-Relationship), EER (Extended Entity-Relationship), and UML (Unified Modeling Language). However, EER provides a more expressive and flexible representation of the relationship between entities in a database than ER and UML[42]. Hence, we have taken EER of a real-time case study as the conceptual model shown in Fig. 4.

**Definition 1** An EER model is defined as $EER = (T, R)$ where $T = \{t_1, \ldots, t_n\}$ is a set of entities, and $R = \{(t_i, t_j) | t_i, t_j \in T\}$ is a set of relationships. A relationship $r = (t_i, t_j)$ represents the mutual connection between entities $t_i, t_j$. Both entities and relationships have a set of attributes.

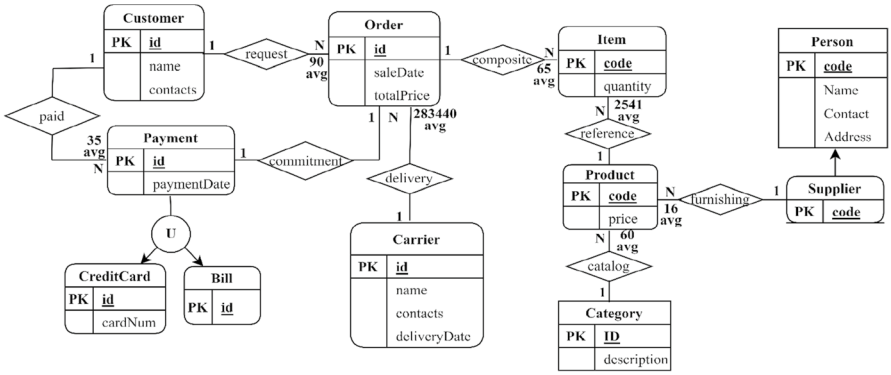


**Fig. 4** The EER schema for an e-commerce application

We have taken a case study adapted from [14], based on the eCommerce domain, as a sample database. The case study states that customers can place orders for various items of different products. Suppliers supply the products and have many categories. Each order has a payment system through credit card or cash. The taken case study is closely related to a real-world scenario, and it's straightforward to explain our work using this sample database. The EER model of the eCommerce case study is shown in Fig. 4. The brief about the EER model is given as follows:

(a) It consists of eleven entities ($T$) named {Person, Category, Customer, Product, Order, Item, Carrier, Supplier, Bill, CreditCard, Payment}
(b) It has eight relationships ($R$) named {request, delivery, owner, reference, composite, catalog, commitment, and furnishing}. Each relationship has its attributes and relationship cardinality (1:1, 1:N, N:1, M: N), indicating how many objects of entities can be associated with objects of another entity.
(c) It also consists of the relationship of special types such as generalization or union. For example, payment consists of two types named Credit Card or Bill. The special types of relationships are treated as regular one-to-one types of relationships.
(d) EER displays the average (avg) access frequencies as estimated data volume by the application users.

We have used the EER of the taken case study to illustrate the work throughout the paper.

### 3.1.2 Application workload

NoSQL databases do not support joins. Embedding or referencing takes the place of the joins in the NoSQL database. The selection between embedding and referencing during data modeling of document stores is the most challenging. Deciding when to embed a document or instead create a reference between separate documents in different collections is an application workload consideration. Additionally,

if the application workload is known during the early data modeling stage, it results in the optimized schema design solution. Hence, the application workload, which includes the estimated database volume and queries, is considered the model's input. The authors have taken the most common seven queries to cover two different scenarios of any eCommerce platform, a) Customer (Q1 and Q2), b) Seller (Q3-Q7). The seven designed queries are shown in Table 2.

### 3.2 Intermediate transformation

The intermediate transformation generates Query Graph (QG) and Query Label (QL). It transforms the application queries into query graphs. The query graphs are used to generate query labels with the help of the application's estimated data volume. We have employed five distinct query labels, OnetoOne Relation, Frequent Lookup, Doc Size, Frequent Modify, and Cardinality, to cover all possible data modeling scenarios [14]. The detailed work of this phase is discussed in this section.

#### 3.2.1 Generate query graphs

Each workload query returns information regarding one or more EER model entities. The derived information from EER is represented as a Query Graph (QG) [43, 44]. A $QG$ is a sub-graph derived from the EER model.

**Definition 2** A Query Graph (QG $\subseteq$ EER) consisting of $(N, E)$ for each query, $q_i \in Q_n$ is defined as follows: nodes N where $N \subseteq T$ corresponds to entities $T$ of EER as mentioned in $q_i$, and edges $E = (n_i, n_j) \subseteq R$ corresponds to a set of relationships $R = (t_i, t_j)$. The procedure to generate a $\text{QG}_i$ for each query $q_i \in Q_n$ is mentioned as follows:

1. List all the entities ($t_n \in T$) in each query $q_i \in Q_n$.
2. For each $q_i \in Q_n$, identify the starting entity $t_i \in T$. Add the entity as a node $n_i$ in $QG_i$. Traverse the relationship $r = (t_i, t_j) \in R$ in the EER model to determine the other entities ($t_j \in t_n$) that are connected to the starting entity ($t_i$). Add the

**Table 2** Workload queries

| Q_No | Query |
|---|---|
| Q1 | Customers place an order having multiple items of different products |
| Q2 | Customers check out orders |
| Q3 | Supplier adds items of a product under a particular category |
| Q4 | Update the details of all orders delivered by a carrier |
| Q5 | Fetch the details of the top-sold products of the year |
| Q6 | Fetch the details of the maximum sold product category-wise |
| Q7 | Fetch the details of customers who purchase a particular product |

relationship as edge ($e$) and the connected entity as $n_j$ to the query graph ($QG_i$) along with the cardinalities of the relationship $r$.

3. If any entity $t_i \in T$ belongs to relationships of special types, such as generalization, adding the entity as a node $n_j$ and the relationship $r = (t_i, t_j) \in R$ as an edge ($e$) connecting nodes $n_i$ and $n_j$ to ($QG_i$). Add cardinality 1:1 on both sides of the edge $e = (n_i, n_j)$.
4. If any of the traversed entities already added to the query graph have relationships with other entities ($t_k \in t_n$), repeat step 2 to traverse these relationships and add the connected entity to the query graph ($QG_i$).
5. Continue this process until all the entities ($t_n \in T$) are traversed along with the relationships.
6. Repeat the above steps for each query $q_i \in Q_n$.

Figure 5 depicts the Query Graphs ($QG$) for all seven input queries mentioned in Table 2.

### 3.3 Generate query labels

In document stores, the entities of the EER model are represented by a collection. In contrast, the documents represent key-value pairs that specify the records

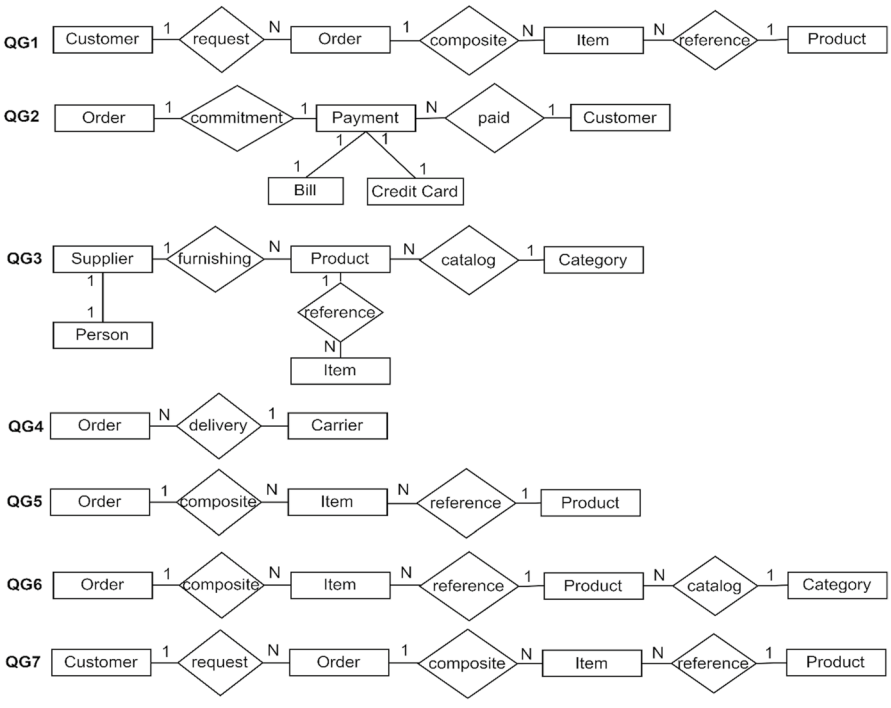


**Fig. 5** Query Graphs generated from workload queries

**Table 3** Embed/ Reference based on the type of relationship

| Type of relationship | Embed/Reference |
|---|---|
| One to one (1:1) | Embed |
| One to many (1: N) or many to one (N:1) | Embed or reference |
| Many to many (M: N) | Embed or reference |

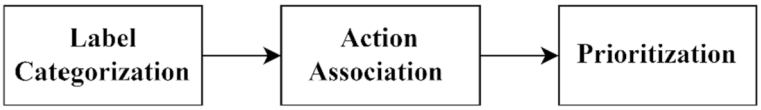


**Fig. 6** Generation of query labels

contained within the entities. The relationships are replaced by embedding or referencing. There are three types of relationships named one-to-one (1:1), one-to-many or many-to-one (1:N or N:1), and many-to-many (M: N) between the entities in the EER model. According to the official documentation of MongoDB [45], as shown in Table 3, only embedding has to be done during data modeling of document stores for a one-to-one (1:1) type of relationship.

However, for the other two types, named one-to-many or many-to-one (1:N or N:1) and many-to-many (M: N) relationships, we either embed the related documents into a single collection or used referencing between distinct documents from different collections. Embedding the documents or making a reference across different collections is an application-specific decision that depends on data growth, read–write ratio, and query types. Based on these factors, the proposed model resolves the trade-off between embedding and referencing in the form of Query Labels (QL). The QG and expected data volume are utilized to determine QL. We have used five labels: OnetoOne Relation, Frequent Lookup, Doc Size, Frequent Modify, and Cardinality (Table 5).

(a) OnetoOne Relation: Each one-to-one relationship belonging to *QG* is labeled as 'OnetoOne Relation. '
(b) Frequent Lookup: If two or more entities are accessed frequently together repeatedly. So, a 'Frequent Lookup' label is added to these entities.
(c) Doc Size: The expected monthly access frequency of entity pairs in the application workload can be used to forecast the future size of a document. If the document size is expected to exceed 16 MB, the 'Doc Size' label is assigned to those entity pairs.
(d) Frequent Modify: When two or more entities are frequently inserted, updated, or deleted, we use 'Frequent Modify' labels on these entities.
(e) Cardinality: If the ratio gap between many-to-many (M: N) types of relationships are high, then use 'Cardinality' labels on these entities.

The process of query label generation is broken down into three steps: Label Categorization, Action Association, and Prioritization, as shown in Fig. 6.

*Step 1* Label Categorization.

NoSQL databases are designed for high performance and scalability, and one of the ways they achieve this is by storing related data together in a single document. NoSQL allows for faster data retrieval, as the data needed for a particular query is more likely to be in a single location. By analyzing the query characteristics, it is possible to determine which entities and attributes need to be accessed together and group them in MongoDB. We have represented the query characteristics in the form of Query Labels (QL). The term "entity pairs" are used in the process of Label Categorization, which refers to the nodes $(n_i, n_j)$ bounded by an edge $(e)$ (relationship) within a query graph. The entity pair represents a specific connection or association between two entities in the Query Graph and are utilized to label the edges based on the relationships among the different nodes.

**Definition 3** For QG $= (N, E)$, an Entity Pair $(n_i, n_j, r)$ is defined as nodes $(n_i, n_j)$ where $(n_i, n_j) \in N$ bounded by a relationship $r \varepsilon E$ is the edge connecting two nodes $(n_i, n_j)$. The procedure to produce all possible entity pairs from $\text{QG}_i | i = 1 \ldots n$ is given below:

1. Initialize an empty list named 'EntityPairs.'
2 For each edge $e \varepsilon E$ in $QG_i$, let $n_i$, $n_j$ be the source and target nodes of $e$. Create an entity pair $(n_i, n_j, r)$ and add the pair to the 'EntityPairs' list.
3. Repeat for each query graph $QG_i$ in the query graph list $(QG_n)$.
4. Return the 'EntityPairs' list, which contains all the unique entity pairs across the query graphs $(QG_i | i = 1 \ldots n)$.

For the eCommerce case study, eight 'Entity pairs (EP)' are formed from QG, as shown in Fig. 7.

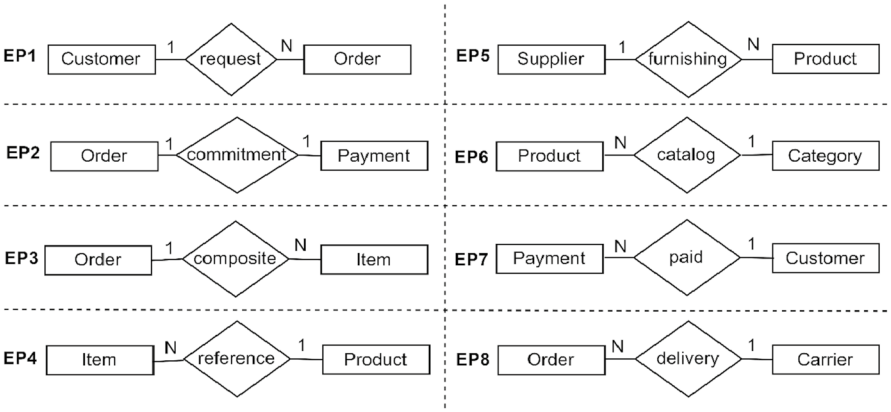


**Fig. 7** Entity pairs (EP) based on query graphs

Along with this, we have taken 40% as the threshold value. This assumption is rooted in the 60–40 rule, which states 60% of the profit involves 40% of the data [46]. This rule helps identify the most critical data and optimize the database's performance by focusing on that data. The details of the Label Categorization are given in Algorithm 1. The theoretical explanation is given as follows:

(a) 'OnetoOne Relation' label is assigned to all the entity pairs having a one-to-one relationship in the query graph $\left(QG_i | i = 1, \ldots, 7\right)$. For instance, entity pairs, Order-Payment, Supplier-Person, Payment-Bill, and Payment-Credit Card have a one-to-one relation for the taken case study. So, the 'OnetoOne Relation' label is assigned to the entity pairs, as shown in Fig. 8.
(b) 'Frequent Lookup' labels are assigned to relationships accessed repeatedly in the application workload. Calculate the access count for each distinct entity pair to assign the label. A threshold value is calculated, which gives the maximum number of entity pairs that can be assigned the 'Frequent Lookup' label. The formula to calculate the threshold is 40% of the maximum count of frequently accessed entity pairs. The access count is set in ascending order, and the 'Frequent Lookup' label is assigned to the number of entities pair whose count is equal to the number obtained from the threshold beginning from the highest value of access count.

For instance, in the preceding case study, we counted the total number of times the distinct entity pairs are accessed together. As shown in Fig. 9, the total count of frequency accessed entity pairs $QG$ is given as Customer-Order, Product-Category is accessed twice, Order-CreditCard, Customer-CreditCard, and Order-Carrier are accessed once each, Order-Item is accessed four times, and Item-Product entities are accessed together five times. According to the formula, the maximum access count of an entity pair is 5, so the threshold (40% of 5) for 5 is 2. Therefore, the label is assigned to the upper two values (4 and 5). Hence, the entity pairs named Order-Item and Item-Product are labeled as 'Frequent Lookup.'

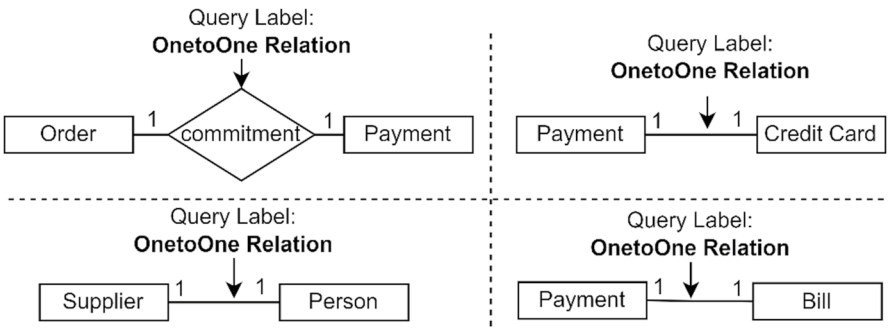


**Fig. 8** 'OnetoOne Relation' Label

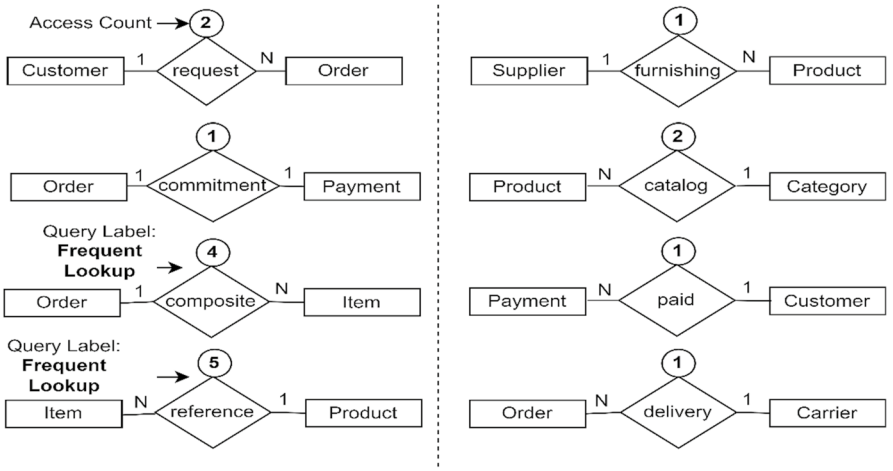


**Fig. 9** 'Frequent Lookup' Label based on Query graphs

(iii) For the 'Doc-Size' label calculation, two values are needed, one is the estimated data volume on each distinct entity pair in the application workload, and the second is the average document size. Among the two, the estimated data volume is taken from the EER model, and the average size of a single document can be calculated with the help of attributes of an entity. But for simplicity, we assume that each document has five attributes (key-value pair) with a maximum size of 12 bytes (the maximum key-value size in MongoDB). Therefore, the average size of each document is 60 bytes. To calculate the 'Doc Size' label, the expected data volume of each distinct entity pair is multiplied by the average document size. From the results, entity pairs with a size of more than 16 MB ($16 \times 10^6$ bytes) are assigned a 'DocSize' label.

As shown in Table 4, for the taken case study, the distinct entity-pair accessed *QG* is named in the first column, and the expected data volume (as mentioned in

**Table 4** The calculation for the 'Doc Size' Label

| Entity pairs from the query graph | Expected access frequency | Expected size in bytes | Expected document size in MB |
|---|---|---|---|
| Order-customer | 90 | 5400 | 0.0054 |
| Order-payment | 1 | 60 | 0.00006 |
| Order-item | 65 | 3900 | 0.0039 |
| Item-product | 2541 | 152,460 | 0.15246 |
| Supplier-product | 16 | 960 | 0.00096 |
| Product-category | 60 | 3600 | 0.0036 |
| Payment-customer | 35 | 2100 | 0.0021 |
| Order-carrier | 283,440 | 17,006,400 | 17.0064 |

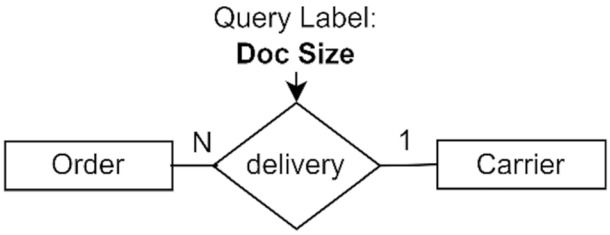


**Fig. 10** 'Doc Size' Label

Fig. 4) for each distinct entity-pair is shown in column 2. The third and fourth column shows the expected document size in bytes and MB, respectively, by multiplying the values of the second column by 60 bytes (taken average size). According to Table 4, the size of the 'Order-Carrier' is more than the threshold value; hence, as shown in Fig. 10, the entity pair is assigned with the 'Doc Size' label.

(d) For 'Frequent Modify' Labels, the queries that perform the database's write (insert, delete, update) operations are selected from the application workload. From the selected workload queries, fetch the corresponding QG. From the selected QG, count distinct entity pairs (related entities) accessed together. The threshold value calculated is 40% of the frequently accessed entity pairs fetched from QG. Then, from the EER model, the data volume of each entity pair is determined. The values of estimated data volume are set in ascending order, and the 'Frequent Lookup' label is assigned to the entity pairs whose count is equal to the threshold beginning from the highest value of estimated data volume count.

Based on the calculations, the 'Frequently Modify' Label is assigned. For instance, queries Q1, Q2, Q3, and Q4 perform the write operations on the database. From the corresponding $QG_i | i = 1, \ldots, 4$ (Fig. 5), Customer-Order, Order-Item, Item-Product, Order-Payment, Supplier-Payment, Product-Category, and Order-Carrier are the total seven entity pairs that are accessed. The threshold value of 7 is 2. Figure 4 shows that the estimated data volume of the above-listed entity pair is given as Order-Customer 90 times, Order-Payment 1 time, Order-Item 65 times, Item-Product 2541 times, Supplier-Payment 16 times, Product-Category 35 times, Order-Carrier 283,440 times. Therefore, the 'Frequently Modify' label is assigned to

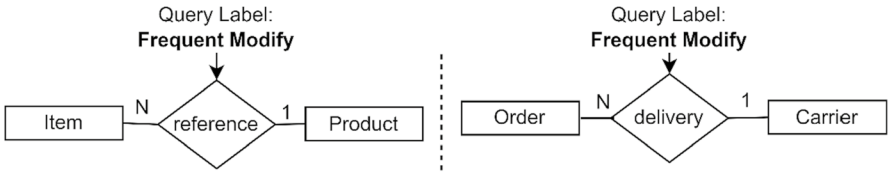


**Fig. 11** 'Frequent Modify' Label

Item-Product and Order-Carrier entity pairs with the uppermost two values, 283,440 and 2541, respectively, as illustrated in Fig. 11.

(e) Cardinality Labels are affixed to many-to-many types of relationships because the ratio of M: N is calculated during logical schema generation of many-to-many types of relationships. If the ratio is high, one-way embedding is performed; otherwise, two-way embedding is performed. The provided case study does not address any M: N type of relationship. But for the sake of understanding, if n consists of a maximum of 5 categories for a book and m consists of a maximum of 50,000 books in a category because the ratio among M:N is high, then apply one-way embedding. If n is a maximum of 3 books written by an author and m consists of a maximum of 5 book authors, the M:N ratio is low, so use two-way embedding.

*Step 2*Action Association.

While schema designing of EER models into MongoDB, embedding or referencing is performed on entity pairs, we have associated the actions to the Query Labels with determining when to embed or refer the entity pairs. The association of actions has been done considering data modeling described on are designed based on the official website of MongoDB [45]. According to the official website, three actions can be performed on entity pairs: One-way Embedding, Two-way Embedding, and Referencing. The action associated with each Query Label is as follows: (i) For the 'OnetoOne' label, the entity pairs must always be embedded together using One-way embedding. (ii) For the 'Frequently Lookup' Label, entity pairs frequently accessed together should always be embedded because it involves too many read operations. Hence the data must be stored at the same location. (iii) For the 'Doc Size' label, the entities should always be referenced because as the document size exceeds 16 MB, MongoDB must allocate a new memory location for the growing document and copy the old document to the new space. It involves many input/output operations and could affect MongoDB's performance. (iv) The 'Frequent Modify' label involves more write operations, including insert, update, and delete operations. The write-intensive entities should always be referenced. (v) Two-way embedding is performed for the 'Cardinality' label, depending on the M: N ratio. The action associated with each label is displayed in Table 5.

*Step 3* Prioritization of Labels.

If a relationship comprises more than one label, label prioritization addresses the trade-off between the actions associated with the labels. According to the summary outlined [6], among embedding and referencing, the highest priority is assigned to referencing because we must first prioritize write-heavy operations and large-size documents. Hence, we have assigned a higher priority, i.e., 1, to reference than embedding. Among one-way and two-way embedding, one-way embedding is assigned priority value 2. In contrast, two-way embedding is assigned priority value 3, as shown in Table 5.

**Table 5** Complete information about Query Labels

| Label name | Description | Label categorization for case study | Associated actions | Label priority |
|---|---|---|---|---|
| OnetoOneRelation | When the related entity pairs have one-to-one (1:1) type of relationship | Person-Supplier, Order-Payment | Use one-way embedding | 2 |
| Frequent lookup | When the related entities are used again and again using the same relationship | Order-Item, Item-Product | Use one-way embedding | 2 |
| Doc size | When the size of a document is expected to be large than 16 MB | Order-Carrier | Use referencing between two entities | 1 |
| Frequent modify | When the entities are frequently updated or deleted | Product-Item, Order-Carrier | Use referencing between two entities | 1 |
| Cardinality | If the cardinality among relationships (M: N) is low | – | Two-way embedding | 3 |

**Algorithm 1: Labels Categorization**

***Input:*** *EER model, Set of Query Graph (QG), Workload Queries*
***Output:*** *Query Labels* ***(QL)***

```
begin
1.  # Generate OnetoOne Relation Label
    for each Query Graph (QGi|i = 1…n)
2.      for each edge e ∈ nj,nk in QGi
3.          if the relationship cardinality on e ∈ nj,nk is 1:1
4.          assign QL= 'OneToOne Relation' on e ∈ ni,nk
5.  # Generate Frequent Modify Label
    for each edge e ∈ nj,nk in QGi
6.      for each Query Graph (QGi) from QGn
7.          count=count+1
8.      m=m+1;
9.      Access[m][0] =nj
10.     Access[m][1] = nk
11.     Access[m][2] = count
12. Sort.Access[m][2] in ascending order
13. Val= 0.4*max * count
14. for (m=0, m<Val, m++)
15.     assign QL= 'Frequent Lookup' to edge e = (Access[m][0], Access[m][1])
16. # Generate Doc Size Label
    for each edge e ∈ nj,nk in QGn
17.     volume=fetch the estimated data volume on the n^th side of rjk≅ejk from EER
        model
18.     size=volume*60*10^-6
19.     if (size> 16)
20.     assign QL= 'Doc Size' to the relationship on entity pair (nj,nk)
21. # Generate Frequent Modify Label
    Query[m]= all write queries from the workload queries (Qn)
22. while (m>0)
23.     for each edge e ∈ nj,nk in QG[m]
24.         count=count+1
25.         volume=fetch the estimated data volume on the nth side of rjk≅ejk from EER
            model
26.         m=m+1;
27.         Access[m][0] =nj
28.         Access[m][1] = nk
29.         Access[m][2] = volume
30.         Sort.Access[m][2] in Ascending order
31. Val= 0.4* max * count
32. for (m=0, m<Val, m++)
33.     assign QL= 'Frequent Modify' to edge e = (Access[m][0], Access[m][1])
34. # Generate Cardinality Label
    for each Query Graph (QGi) from QGn
35.     for each edge e ∈ nj,nk in QGi
36.         if the relationship cardinality on e ∈ nj,nk is M: N
37.         assign QL= 'Cardinality' to the edge e ∈ nj,nk
38. return QL
end
```

### 3.3.1 Final schema generation

Final schema generation is further categorized into two parts: (i) generation of Schema Graph (SG) and Label assignment, and (ii) Transformation into Logical Schema, as shown in Fig. 12. In the generation of the Schema Graph (SG), the EER model is converted into a graph named Schema Graph (SG), and then QL are assigned onto SG. Based on the defined rules, SG is transformed into MongoDB logical schema.

**3.3.1.1 Generation of schema graph and label assignment** The EER model is first converted into a graph model (SG) by representing EER entities as nodes and relationships as edges. After that, the Query Labels (QL) are assigned to Schema Graph (SG).

**Definition 4** A Schema Graph (SG) = $(N_G, E_G)$ can be represented with the help of nodes ($N_G \in T$) and edges ($E_G \in R$). Nodes ($N_G$) must always equal the number of entities ($T$), and Edges ($E_G$) must equal the number of relationships ($R$) in the EER model.

***Algorithm 2*** gives the detailed procedure of Schema Graph (SG) generation. The algorithm iterates through the entities (T) and relationships (R) in the EER model. For each entity ($t \varepsilon T$), a corresponding node ($n_G$) is created in the SG. Similarly, for each relationship ($r \varepsilon R$), an edge ($e_G$) is added to the SG, connecting the start ($t_i$) and end entities ($t_j$) of the relationship ($r$). This process continues until all entities and relationships in the EER model have been processed. The resulting Schema Graph provides a visual representation of the EER model. Following creating the Schema Graph, the query labels are assigned on the edges of SG. Figure 13 depicts the SG generated from the EER model, consisting of 11 nodes and 11 edges along with the assigned QL on edges.

**Algorithm 2: Generation of Schema Graph (SG)**

***Input:*** *EER model (Entities (T), Relationships (R))*

***Output:*** SchemaGraph (SG) *with nodes (*$\mathsf{N_G}$*) and edges (*$\mathsf{E_G}$*)*

```
1. for each entity (tεT) in the EER model
2.     add the entity as a node (nG) in SG
3. for each relationship (rεR) in EER model
4. {
5.     identify the start entity (ti) and end entity (tj) of the relationship (r).
6.     create an Edge (eG) connecting the start node (nGi ≡ ti) and end entity (nGj ≡ tj) in SG
7. }
8. Return SG
```

**3.3.1.2 Transformation into logical schema** A logical schema of document stores is derived from SG. The following rules are followed to generate a logical schema:

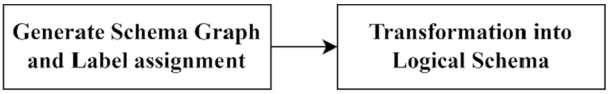


**Fig. 12** Final schema generation

(1) If a single QL is assigned on the edges of SG, the decision is based on the action associated with QL, as discussed in Sect. 3.2.2.
(2) If more than one QL is assigned on the edges of SG, the decision is based on the priority associated with QL, and based on the priority, action must be taken.
(3) If there is no QL between the edges of SG, create a separate collection for each entity in SG.

Figure 14 depicts the final logical schema model for the case study after applying all phases of the proposed model. Since no label is assigned between entity pairs named Supplier-Product, Product-Category, Order-Customer, and Carrier-Customer following rule 3, separate collections are created for each entity. For the rest of the entities, there is a label between the edges, so by following rules 1 and 2, the entities are embedded or referenced among each other.

The procedure of the proposed model that encompasses all the phases is mentioned as Algorithm 3. The proposed algorithm designs the schema automatically by transforming the model's inputs into the logical schema of MongoDB. The Query Graph (QG) is generated for each query $(q_i \in Q_n)$ $q_i \in Q$ of the application workload (Line 1). The Query Labels (QL) is generated using Query Graph (QG) (Line 2). The EER model is converted into Schema Graph (SG) (Line 3) using

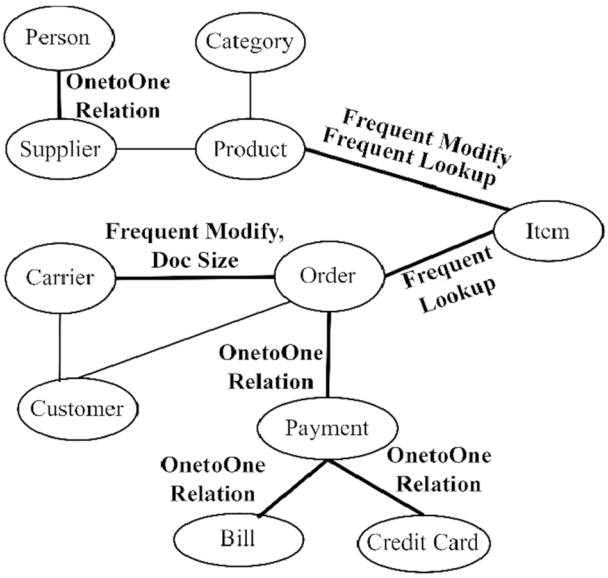


**Fig. 13** Schema Graph (SG) with query labels

Algorithm 2. Then, the calculated Query Label (QL) is assigned to the Schema Graph (SG) (Line 4). The (SG) is converted into the logical schema using actions performed on the assigned Query Label (QL) (Line 5–23). Remove edge $e_i \in E$ from (SG). If it contains any (QL), if it is OnetoOne Relation, embed entities among one another. If the label is Frequent Lookup, for relationship type 1:1 or 1: N, embed the child entity into the parent entity; if the relationship is N:1, then embed the child into the parent as an array of embedded objects. For Doc Size or Frequent Modify label, perform referencing and refer child entity into parent entity. For Cardinality Label, perform one-way or two-way embedding depending on the ratio gap among M: N. If an edge ($e_i \in E$) has more than two query labels and performs actions based on the priority of labels. If no label is assigned to the edge, make a new collection. If the number of entities is N and the number of relationships is M, then Algorithm 3 has a time complexity of $O(N + M)$.

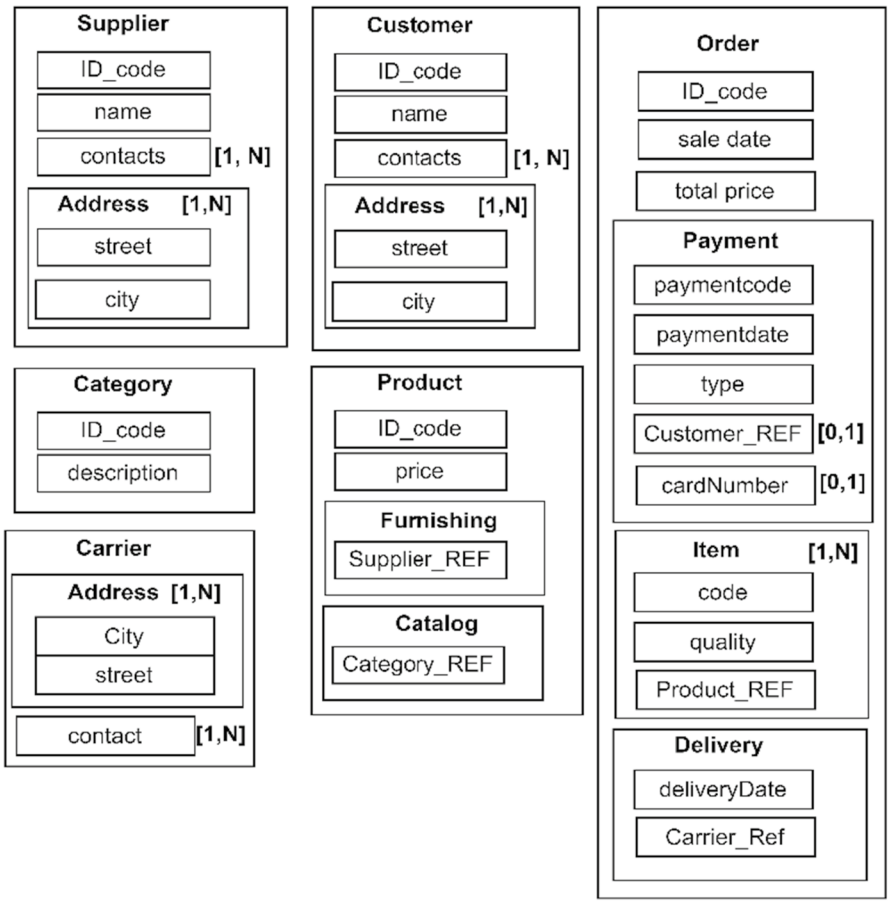


**Fig. 14** Logical schema generated from Schema Graph (SG)

**Algorithm 3: Transform the EER model to MongoDB logical schema**

***Input:*** *The entities and relationships of EER model, Application queries, Data Volume*

***Output:*** *The logical schema design of MongoDB*

***begin***

1. Make **QueryGraph (QG)** for each query $\mathbf{q_i}$ in workload.
2. Generate **QueryLabel (QL)** based on **QueryGraph (QG)** and expected data volume /*refer algorithm 1*/
3. **SchemaGraph(SG)** ← ConvertToGraph (EER model) /*refer algorithm 2*/
4. **Schema_Graph (SG)** ← add **QueryLabel (QL)**
5. **for** each edge ($e_i$|i=1…n) in **SG** do
6. **if** $\mathbf{e_i}$ has one **QueryLabels (QL)**
7. **do**
8. **if QueryLabel(QL) == OnetoOne Relation**
9. Embed child entity to parent entity as an object
10. **else if QueryLabel(QL) == Frequent Lookup**
11. **switch** Type of relationship do
12. **case 1**: (**OneToOne or OneToMany**)
13. Embed child entity to parent entity as an object
14. **case2**: (**ManyToOne)**
15. Embed as an array from parent entity to child entity
16. **else If QueryLabel (QL) == (Doc Size or Frequent Modify**)
17. Refer the parent entity into the child entity
18. **else QueryLabel (QL) == (Cardinality**) // Two way embedding
19. refer child entity to parent entity as an array and refer parent entity to child entity as an array
20. **else if** $\mathbf{e_i}$ **has more than one QueryLabels (QL)**
21. **if QueryLabel (QL) == (Doc Size and Frequent Modify**)
22. Jump step 17
23. **if QueryLabel (QL) == (OnetoOne and Frequent Lookup**)
24. Jump step 11
25. If **QueryLabel (QL) == ((OnetoOne or Frequent Lookup or Cardinality) and (Doc Size or Frequent Modify))**
26. Jump step 17
27. If **QueryLabel (QL) == ((OnetoOne or Frequent Lookup) and (Cardinality))**
28. Jump step 11
29. **else QueryLabel (QL)** does not exist
30. Make two separate collections for each entity and refer one to another
31. **end for**

**end**

## 4 Experimental evaluation

We evaluate our approach with an experiment in the e-commerce domain, as described in Sect. 3. The experiments are conducted to validate our proposed model and demonstrate the model's positive effects on query processing time. The performance of our proposed model is compared with three existing models: (i) *Conventional* [37]: It is workload agnostic, and logical schema is designed without taking an application workload by following the relationship constraints only as shown in Fig. 15a; (ii) *Optimized* [37]: As shown in Fig. 15b, it is workload-driven but logical schema design, is generated based solely on expected data volume of application workload, and (iii) *Query Path Graph* (*QPG)* [17]: It considers application query patterns for the data modeling transformation of

conceptual to logical modeling as shown in Fig. 15c. We chose these three existing models because they define and explore three distinct ways of MongoDB data modeling, and their work is comparable to our proposed work. Table 6 gives the qualitative analysis of the four models based on various important factors named Query Response Time, Query Speedup, Write Latency, Read Latency, Number of Pipeline Stages, Pipeline efficiency, Storage Space, Scalability, Throughput and Latency. It has been found that our model works well for all factors. However, experiments are conducted to prove the qualitative analysis in numbers. The following section details the experimental setup and results to verify the qualitative analysis.

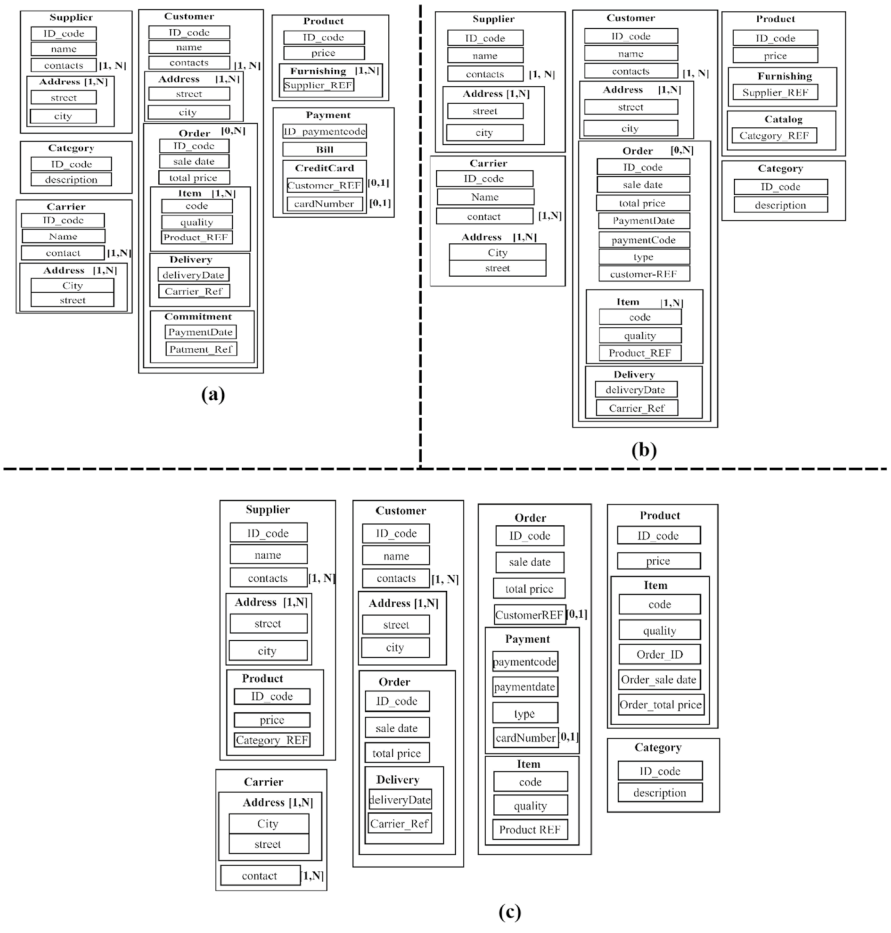


**Fig. 15** Existing Logical model used for performance comparison **a** Conventional model, **b** Optimized, **c** QPG

### 4.1 Experimental setup

The experimental analysis is performed using an Intel Core i7- 1255U processor with 16 GB of RAM, 3-level cache, and 1 TB hard disk. The data is stored using MongoDB Atlas, a cloud-based database service provided by MongoDB. The experimental setup utilized a MongoDB Atlas Cluster M20 configuration dedicated to 4 GB RAM with 20 GB storage and 2vCPUs. The cluster is configured on AWS/ Mumbai (ap-south-1) region running MongoDB version 6.0.6. It consisted of a replica set with three nodes to ensure load balancing of read and write operations, data availability, and reduced query response time. Studio 3 T, a graphical user interface (GUI) based MongoDB IDE [47], is also used. The experimental setup information is summarized in Table 7.

To make the query performance comparison, we created four physical databases for schema shown in Figs. 14 and 15 in MongoDB and named as conventional, optimized, QPG, and proposed. We have populated all four databases with identical data, as shown in Table 8. The seven queries (Q1-Q4 CRUD queries, Q5-Q7 aggregate pipeline) outlined in Sect. 3.2 are executed in each database. As stated in Table 9, along with the seven queries taken as input, the performance is measured on eight additional queries (Q8-Q15) because queries evolve with time, and new queries are always added to the system. Hence, to measure the performance of a system, it is necessary to measure it on run-time queries.

### 4.2 Experimental evaluation

To perform the experimental analysis, we have measured various essential performance parameters of a database as listed in Table 6. Based on the parameters, the following comparison is made among proposed and existing models: (1) Query Response Time and Speedup (Sect. 4.2.1), (2) Read and Write Latency (Sect. 4.2.2), (3) Efficiency Improvement using aggregate pipeline (Sect. 4.2.3), (4) Storage

**Table 6** Qualitative analysis of existing and proposed model

| Srl. No | Factors | Conventional | Optimized | QPG | Proposed |
|---|---|---|---|---|---|
| 1 | Query response time | High | High | High | Low |
| 2 | Query speedup | Slow | Slow | Very slow | Fast |
| 3 | Write latency | High | High | Very high | Low |
| 4 | Read latency | High | High | Very high | Low |
| 5 | No. of pipeline stages | Very high | High | High | Low |
| 6 | Pipeline efficiency | Low | Low | Low | High |
| 7 | Storage space | Less | Less | High | Least |
| 8 | Collection-wise performance | Good | Good | Worst | Best |
| 9 | Scalability | Efficient | Efficient | Poor | Most efficient |
| 10 | Throughput | Low | Moderate | Low | High |
| 11 | Latency | Low | Low | Low | Low |

**Table 7** Experimental setup

| Hardware configuration | |
|---|---|
| Processor | Intel Core i7- 1255U running at 1.90 GHZ |
| RAM | 16 GB |
| Cache | 3 Level |
| Storage | 256 MB (SSD), 1 TB hard-disk |
| Software configuration | |
| Database service | MongoDB Atlas |
| Cluster configuration | Cluster M20 |
| Cluster RAM | 4 GB |
| Cluster storage | 20 GB |
| Cluster vCPUs | 2 |
| Network configuration | AWS / Mumbai (ap-south-1) |
| MongoDB version | 6.0.6 |
| Replica set nodes | 3 |

**Table 8** A eCommerce dataset, along with the number of records in each EER table

| | List of EER tables | Number of records in each table |
|---|---|---|
| Dataset | Person | 100 |
| | Order | 50,000 |
| | Carrier | 16 |
| | Supplier | 149 |
| | Customer | 10,000 |
| | Product | 400 |
| | Item | 70,000 |
| | Bill | 20,000 |
| | Credit card | 10,000 |
| | Payment | 30,000 |
| | Category | 60 |

**Table 9** Additional run-time queries for performance evaluation

| Q_No | Query |
|---|---|
| Q8 | Given an order id, return the order and related customer, items and products |
| Q9 | Given an order id, return the order and related customer and payments |
| Q10 | Given a customer id, return all orders and related carriers |
| Q11 | Given a customer id, return all orders and related payments |
| Q12 | Given a product id, return all related items and orders |
| Q13 | Given a supplier id, return the supplier and all related products, including their categories |
| Q14 | Add new attributes, city, and country in the Customer Information |
| Q15 | Add a new category of the product |

Space (Sect. 4.2.4), (5) Collection-wise Performance (Sect. 4.2.5), (6) Scalability (Sect. 4.2.6), (7) Throughput and Latency (Sect. 4.2.7).

### 4.2.1 Query response time and speedup

It refers to the time the database takes to process and respond to a request for information. Query response time is an important performance parameter, as it can affect the speed and efficiency of the database. In general, faster query response times are desirable, as they can lead to better performance and user experience. Also, we have calculated a unitless speedup factor calculated by the mean query response time for the proposed model divided by the mean query response time for the existing model. The overall result is summarized through the Geometric Mean (GM) of all 15 queries. Because GM is the appropriate, meaningful average for normalized unitless numbers [48]. GM helps make broad at-a-glance speedup comparisons among existing and proposed model's performance. The formula to calculate the query speedup factor is:

$$\text{Average Query Speedup Factor} = \sqrt[N]{{}_{n=1}{}^{N}\Pi Q_n}$$

where $Q_n$is speed up factor for each query$q_i$

$$Q_n = \frac{T_{i_P}}{T_{i_{\text{EM}}}}$$

where $T_{i_P}$ = Query execution time of ith query of proposed model, $T_{i_{\text{EM}}}$ = Query execution time of ith query of existing model, $N$ = total number of workload queries.

We have performed an extensive evaluation of proposed models against existing models on MongoDB. We ran each of the 15 queries on four models to conduct the experiment. Three runs for each query were made to avoid the distorted results by caches in MongoDB. The average value of three runs is taken as query response time. The speedup factor for an individual query is then calculated by dividing the particular query ($q_i \in Q_n$, where$i = 1, \ldots, 15$) response time of the existing model by the query response time of the proposed model. Table 10 details the complete numerical figures of query response time and the speedup factor for all 15 queries, whereas the graphical representation of query response time is shown in Fig. 16. The average speedup factor is shown in the last row of Table 10. The following observations are made from Fig. 16.

It can be observed that the response time of the proposed model for queries Q1 and Q2 is less time than all other models. For queries Q5, Q6, Q8, and Q9, the proposed model performs much better than the existing models. The performance improvement is due to the reason that these queries access a specific Order, including Items. The conventional and optimized schema nests the Orders inside the Customer collection, whereas QPG has to make reference for the Orders with other collections named Items which is very time-consuming. In contrast, the schema generated by our model reads the documents from the Order collection directly. For queries Q7, Q10, and Q13 proposed model is better than QPG but performs

**Table 10** Query execution time of workload queries

| Q.No | Query operation | Query execution time (milliseconds) | | | | Speedup factor for each query | | |
|---|---|---|---|---|---|---|---|---|
| | | Conventional (C) | Optimized (O) | QPG | Proposed (P) | C versus P | O versus P | QPG versus P |
| Q1 | Write | 191.5 | 182.0 | 154.0 | 101.0 | 1.9 | 1.8 | 1.5 |
| Q2 | Write | 305.0 | 295.5 | 282.5 | 175.5 | 1.7 | 1.7 | 1.6 |
| Q3 | Write | 241.7 | 271.2 | 391.8 | 281.2 | 0.9 | 1.0 | 1.4 |
| Q4 | Write | 204.2 | 193.5 | 172.7 | 182.3 | 1.1 | 1.1 | 0.9 |
| Q5 | Read | 6764.0 | 5345.3 | 7477.0 | 4019.3 | 1.7 | 1.3 | 1.9 |
| Q6 | Read | 12,737.0 | 13,463.3 | 15,317.3 | 10,884.0 | 1.2 | 1.2 | 1.4 |
| Q7 | Read | 3718.3 | 3808.7 | 4909.3 | 4076.0 | 0.9 | 0.9 | 1.2 |
| Q8 | Read | 1311.3 | 1153.0 | 1442.7 | 911.3 | 1.4 | 1.3 | 1.6 |
| Q9 | Read | 901.0 | 1063.0 | 1029.3 | 876.0 | 1.0 | 1.2 | 1.2 |
| Q10 | Read | 2081.3 | 2797.0 | 4852.0 | 3452.0 | 0.6 | 0.8 | 1.4 |
| Q11 | Read | 3698.7 | 974.0 | 1675.7 | 1842.3 | 2.0 | 0.5 | 0.9 |
| Q12 | Read | 1277.7 | 1444.3 | 1329.3 | 1162.7 | 1.1 | 1.2 | 1.1 |
| Q13 | Read | 2230.3 | 1871.3 | 1158.7 | 1858.7 | 1.2 | 1.0 | 0.6 |
| Q14 | Write | 322.3 | 341.5 | 391.5 | 291.3 | 1.1 | 1.2 | 1.3 |
| Q15 | Write | 101.0 | 115.7 | 205.0 | 161.7 | 0.6 | 0.7 | 1.3 |
| Speedup factor | | | | | | 1.2 | 1.1 | 1.3 |

poorly than conventional or optimized because the queries access Customer records, including Orders. In both conventional and optimized, the information can be directly accessed from the customer-rooted collection. But in QPG and the Proposed model, the Customer collection has to be linked with the Order collection. However, in QPG, more time is taken because orders are nested inside customers and have a separate collection. So, time is taken to perform both nesting and referencing from customer to order collection. For Q11, the performance of the proposed model is better than conventional but poor than optimized and QPG models, while Q12 performs poorly than all three existing models. For Q4, Q14, and Q15, all three models have almost the same performance. Hence, we can conclude that the performance of the proposed model is similar to or better than existing models for both input and run-time queries.

### 4.2.2 Write and read latency

Read latency is the time taken to retrieve data from the database, while write latency is the time taken to store data in the database. Latency can be affected by factors such as the schema from which data is being read or written, the workload on the database, and the type of storage used. In our case, to measure the effect on latency due to schema and workload, the queries are divided into two categories named (1) Write queries (Q1-Q4, Q14-Q15), and (2) Read queries (Q5-Q13). We have taken the average query response time for all underlying categories. The resultant table is shown in Table 11, while the graphical representation is shown in Fig. 17. It shows that the proposed model reduces the write latency by a factor of 1.14, 1.17, and 1.33 while read latency by a factor of 1.19, 1.09, and 1.37 than Conventional, Optimized, and QPG, respectively. Hence, the proposed model outperforms all three existing models regarding write and read latency, as the lowest latency means better performance.

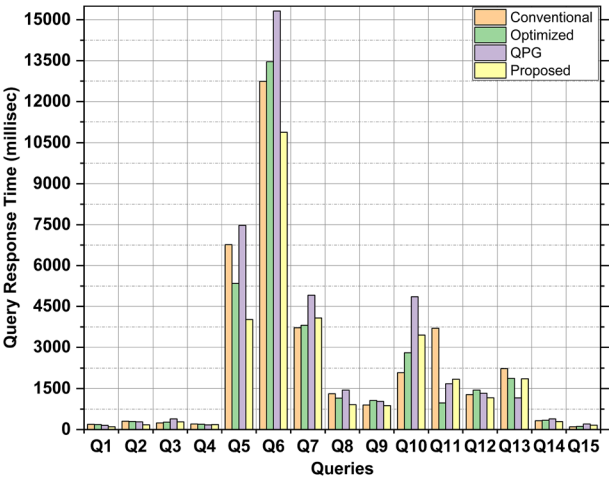


**Fig. 16** Query response time comparison among different

**Table 11** Write and Read latency for each schema model

| Query operation | Conventional (C) | Optimized (O) | QPG | Proposed (P) | Speedup factor | | |
|---|---|---|---|---|---|---|---|
| | | | | | C versus P | O versus P | QPG |
| Write | 1365.7 | 1399.3 | 1597.5 | 1193.0 | 1.14 | 1.17 | 1.33 |
| Read | 34,719.7 | 31,920.0 | 39,191.3 | 29,082.3 | 1.19 | 1.09 | 1.37 |

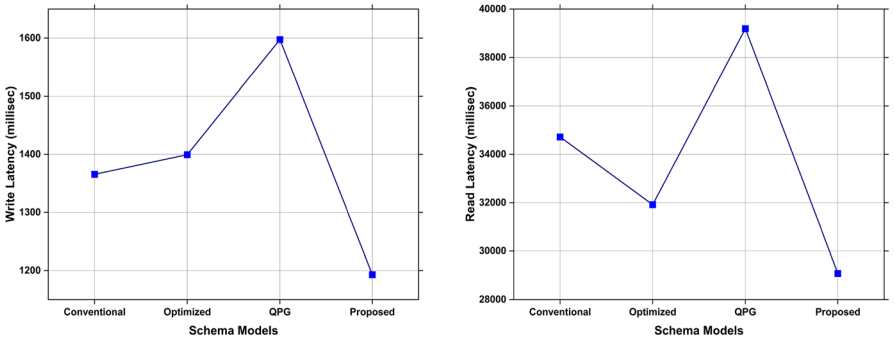


**Fig. 17** Write and Read Latency among Proposed and existing models

### 4.2.3 Efficiency improvement using aggregate pipeline

MongoDB uses an aggregate pipeline framework for complex query processing [49]. The aggregate pipeline processes documents in different collections and returns computed results. An aggregate pipeline is a powerful tool for data processing and analysis in MongoDB. It can perform a wide variety of operations on data, including data transformation, data aggregation, and data analysis.

The efficiency ($\eta$) of the aggregate pipeline in MongoDB depends on several factors, including the number of pipeline stages and the size and organization of the data being processed. Efficiency improvement ($\eta\%$) measures the performance of the aggregate pipeline with a minimum number of aggregate stages. To calculate ($\eta$), firstly, the execution plan is analyzed, which shows the total number of stages for MongoDB individual query for all four models, as shown in Table 12. Then the total number of stages ($S = \sum_{i=1}^{n} S_i$) for each query $q_i \in Q_n$ where $i = 1, \ldots, 15$ is calculated. The formula used to calculate the efficiency improvement is:

$$\text{Efficiency improvement } (\eta\%) = (\eta - 100)\%, \text{ where } \eta = \frac{S_{\text{EM}_i}}{S_P} * 100 | i = 1, 2, 3$$

where $S_{\text{EM}_i}$ = Total number of stages in $i$thexistingmodel, $S_P$ = Total number of stages in proposed model

As mentioned above, we have nine aggregate queries among a total of fifteen queries. Hence, we have taken those aggregate queries to analyze the efficiency improvement. Table 12 shows the total number of stages used by each query and the total number of stages used by all nine queries. The efficiency improvement of the

aggregate pipeline of the proposed method against existing models is calculated by the formula mentioned above.

Table 13 illustrates the percentage efficiency improvement of the proposed model against the conventional model is given by 17.5%. In contrast, for the optimized model, it is given as 15%; for the QPG model, it is given as 10%. This section concludes our method provides better pipeline efficiency than the existing methods. The reason for better efficiency is that the aggregate pipeline stages depend on the number of documents fetched against the application query. A schema designed based on the application queries results in fewer documents being scanned during query processing. Hence, we can conclude that our model suggests the best logical schema for the application workload than the existing models.

#### 4.2.4 Storage space

In MongoDB, the size of a collection refers to the total amount of disk space consumed by the data within that specific collection. The storage requirements in MongoDB can vary greatly depending on factors, such as data volume, data model, and query usage patterns. Due to these factors, the various models, including the Conventional, Optimized, QPG, and Proposed, introduce different collections and exhibit variations in storage space. Table 14 shows each model's storage space and the total number of documents in each collection. Due to the flexibility property, the four models (Conventional, Optimized, QPG, and Proposed) have different collections, and a collection in one model may or may not be present in another model. For example, the "Order" collection is only provided in the QPG and Proposed models, whereas the "Payment" collection is only in the conventional model.

Figure 18 represents the graphical visualization of storage space variations across different models. Figure 18a illustrates the space occupied by each collection individually, providing a collection-wise comparison among the models. On the other hand, Fig. 18b shows the total storage space occupied by all the collections within a specific model.

Figure 18a shows that all four models have the same disk space for the Carrier, Supplier, and Category collections, indicating that these collections have almost consistent storage requirements across the models.

1. The disk space occupied by the Customer collection gradually decreases from the Conventional model to the Proposed model, indicating storage optimizations in the latter models. This decrease in disk space is caused by a lower level of embedding in the proposed models when compared to the Conventional model.
2. The Product collection shows variations in disk space among the different models. Compared to the Conventional and QPG models, the Optimized and Proposed models have more disk space allocated for the Product collection. This difference in disk space can be attributed to the fact that the Optimized and Proposed models include Furnishing and Catalog embedded documents within the Product collection. These additional embedded documents contribute to the Optimized and Proposed models' higher disk space utilization when compared to the other models.

**Table 12** Number of Pipeline stages for each aggregate query

| Q# | Conventional (C) | | Optimized (O) | | QPG | | Proposed (P) | |
|---|---|---|---|---|---|---|---|---|
| | PS | TC | PS | TC | PS | TC | PS | TC |
| Q5 | Unwind, Project, Unwind, Group, Group | 5 | Unwind, Project, Unwind, Group | 4 | Unwind, Project, Unwind, Group, Group | 5 | Unwind, Group, Project | 3 |
| Q6 | Unwind, Project, Group, Lookup, Unwind, Project, Lookup | 7 | Unwind, Project, Unwind, Group, Lookup, Unwind, Project, Lookup | 8 | Unwind, Project Group, Unwind Lookup | 5 | Unwind, Group, Project, Group | 4 |
| Q7 | Unwind, Project, Unwind, Match, Group | 5 | Unwind, Project, Match, Group | 4 | Unwind, Project, Match, Group | 4 | Unwind, Match, Project | 3 |
| Q8 | Match, Unwind, Unwind, Lookup, Group | 5 | Match, Unwind, Unwind, Lookup, Unwind, Group | 6 | Match, Lookup, Unwind, Unwind, Lookup, Unwind, Group | 7 | Match, Lookup, Unwind, Unwind, Lookup, Unwind, Group | 7 |
| Q9 | Match, Unwind, Lookup, Unwind, Group | 5 | Match, Unwind, Unwind, Lookup, Unwind, Group | 6 | Match, Lookup, Unwind | 3 | Match, Lookup, Unwind, Unwind, Lookup, Unwind, Group | 7 |
| Q10 | Match, Unwind, Lookup, Unwind, Unwind, Group | 6 | Match, Unwind, Lookup, Unwind, Group | 5 | Match, Unwind, Lookup, Unwind, Lookup, Group | 6 | Match, Lookup | 2 |
| Q11 | Match, Unwind, Lookup, Group | 4 | Match | 1 | Match, Unwind, Lookup, Unwind, Group | 5 | Match, Lookup | 2 |
| Q12 | Match, Lookup, Unwind, Unwind Group | 5 | Match, Lookup, Unwind, Unwind, Match, Group | 6 | Match, Lookup, Lookup, Unwind | 4 | Match, Lookup, Unwind, Unwind, Match, Group | 6 |
| Q13 | Match, Lookup Unwind, Lookup Group | 5 | Match, Lookup, Unwind, Lookup Unwind, Lookup | 6 | Match, Lookup, Unwind, Lookup, Unwind | 5 | Match, Lookup, Unwind, Lookup, Unwind, Group | 6 |
| Total | | 47 | | 46 | | 44 | | 40 |

*PS* Pipeline stages, *TC* Total count, *C* Conventional, *O* Optimized, *P* Proposed

**Table 13** Aggregate pipeline efficiency of the proposed model against existing models

| | C versus P | O versus P | QPG versus P |
|---|---|---|---|
| Total pipeline stages (PS) | 47/40 | 46/40 | 44/40 |
| η% | 17.5% | 15% | 10% |

3. The Order collection is absent in the Conventional and Optimized models but present in the QPG and Proposed models, which have significantly more disk space.
4. 17b shows that the total disk space for Conventional, Optimized, and Proposed models is relatively similar, with the Proposed model showing a slight reduction. The QPG model consumes more total disk space than the other three models, owing to the repetition of the Order collection separately and within the customer collection, which takes up a significant amount of disk space.

#### 4.2.5 Collection-wise performance

Collection-level performance analysis using "MongoTop" is a valuable technique to track the time taken by read and write activity of each collection in a MongoDB instance. The benefit of this is that it provides insights into the most active collections regarding disk I/O operations, which can help identify performance bottlenecks and optimize database operations. "MongoTop" is a tool provided by MongoDB that continuously samples data over a specified duration and provides real-time reports on the activity of individual collections. We have analyzed three important parameters (Total, Read, Write) to gain insights into a deeper understanding of Collection-wise performance. The "Total" shows the total amount of time, in microseconds, spent performing both read and write operations on a particular collection. By examining this metric, we can assess a collection's overall workload and activity level. The "Read" parameter indicates the amount of time, in microseconds, desiccated to read operations on a particular collection. The benefits of analyzing this metric are to identify heavily read-intensive collections, providing insights into the data access patterns and usage characteristics. The "Write" parameter displays the amount of time, in microseconds, spent on performing write operations on a collection. By examining this metric, we can identify collections that experience significant write activity, enabling us to focus on optimizing write-intensive operations. Table 15 provides information on the Total, Read and Write activity for each collection across the Conventional, Optimized, QPG, and Proposed models.

Figure 19 shows the graphical representation of information shown in Table 15. Figure 19 has three parts: 19a, 19b, and 19c. Part '19a' compares the four models (Conventional, Optimized, QPG, and Proposed) based on the total time spent on read-and-write operations for each collection. Part '19b' compares different collections of a model based on Read time, whereas '19c' highlights the write time for each collection among four models. A detailed explanation is given below:

**Table 14** Storage space occupied by each collection, along with total space occupied by each model

| Collection name | Conventional | | Optimized | | QPG | | Proposed | |
|---|---|---|---|---|---|---|---|---|
| | Total documents | Disk space (MB) | Total documents | Disk space (MB) | Total documents | Disk space (MB) | Total documents | Disk space (MB) |
| Carrier | 60,008 | 12.30 | 60,008 | 12.30 | 60,008 | 12.30 | 60,008 | 12.30 |
| Category | 60 | 0.94 | 60 | 0.94 | 60 | 0.94 | 60 | 0.94 |
| Customer | 415,567 | 464.55 | 415,243 | 458.31 | 405,456 | 425.55 | 150,023 | 25.96 |
| Product | 112,850 | 9.10 | 115,830 | 11.93 | 90,119 | 7.91 | 115,830 | 11.93 |
| Supplier | 60,110 | 12.32 | 60,110 | 12.32 | 61,456 | 21.42 | 60,110 | 12.32 |
| Payment | 26,100 | 2.76 | – | – | – | – | – | – |
| Order | – | – | – | – | 762,880 | 509.03 | 562,880 | 429.03 |
| Total storage (MB) | 501.96 | | 495.79 | | 977.14 | | 492.47 | |

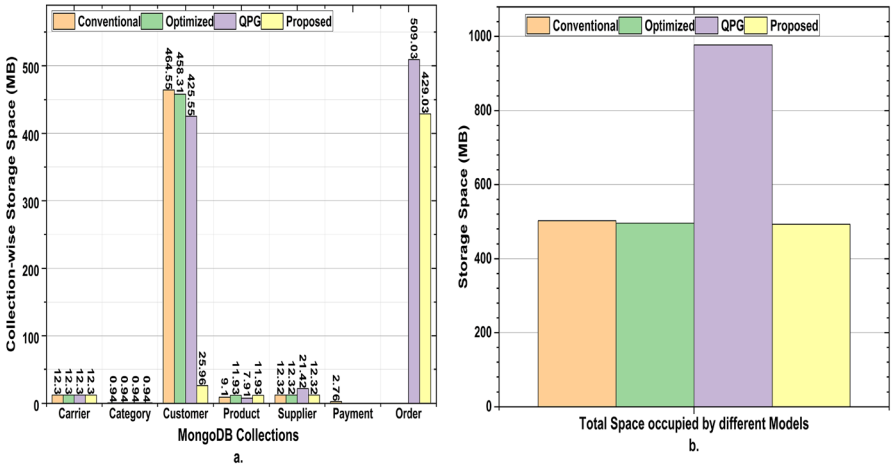


**Fig. 18** Storage Space comparison **a** Collection-wise storage comparison **b** Total space occupied by different models

1. Figure 19a shows the proposed model has the lowest total time across all collections compared to the other models. The collections in Optimized and Conventional models show slightly higher total times compared to the Proposed model but remain relatively close. The QPG model stands out with significantly higher total times, primarily due to the Order collection, which substantially impacts the overall time.
2. Figure 19b shows Customer collection; the Conventional model has the highest read time, followed by the Optimized model. The QPG and Proposed models have significantly lower read times. Therefore, the overall performance of the Conventional and Optimized models depends on the Customer collection only. The Proposed model generally shows reduced read times compared to the Conventional and Optimized models, suggesting improved read performance. However, the QPG model exhibits higher read times for certain collections due to the Order collection in multiple locations.
3. Figure 19c shows the QPG model has the highest write time for the Customer collection, while the other models have relatively lower write times. Also, the Order collection has a high time in QPG. The Proposed model generally demonstrates lower write times than the Conventional and Optimized models. Notably, among the four, the proposed model performs better than all existing models in terms of write operations.

The graphs highlight the performance differences among the Conventional, Optimized, QPG, and Proposed models. The Conventional and Optimized models exhibit similar total, read, and write times, which are higher than those of the

**Table 15** Collection-wise performance comparison

| Collection name | Conventional | | | Optimized | | | QPG | | | Proposed | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | Read | Write | Total | Read | Write | Total | Read | Write | Total | Read | Write |
| Carrier | 166 | 166 | 0 | 179 | 79 | 0 | 180 | 180 | 0 | 179 | 179 | 0 |
| Category | 6331 | 6279 | 52 | 5655 | 5595 | 60 | 4062 | 3993 | 69 | 4085 | 4032 | 53 |
| Customer | 25,420 | 21,999 | 3421 | 22,355 | 18,569 | 3786 | 10,056 | 5711 | 4345 | 8047 | 5706 | 2341 |
| Product | 2580 | 2008 | 572 | 3785 | 3414 | 371 | 5869 | 4835 | 1034 | 2553 | 2097 | 456 |
| Supplier | 1434 | 1434 | 0 | 1345 | 0 | 0 | 2567 | 2567 | 0 | 1467 | 1467 | 0 |
| Payment | 154 | 154 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Order | 0 | 0 | 0 | 0 | 0 | 0 | 18,054 | 13,602 | 4452 | 13,944 | 13,155 | 0 |

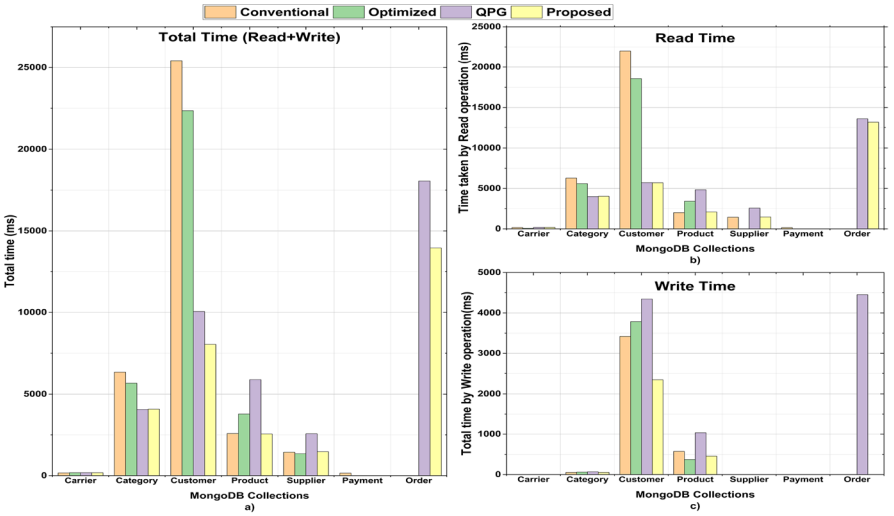


**Fig. 19** Collection-wise performance comparison **a** Time taken by both read and write operations, **b** Time taken by read operations, and **c** Time taken by both read operations

Proposed model. This is because the Customer collection in both models is heavily embedded with Order documents, leading to performance bottlenecks. In contrast, the Proposed model addresses this issue by separating the Customer and Order collections, resulting in improved performance. The QPG model stands out with significantly higher write and read times, attributed to the repetitive Order collection. Overall, the Proposed model consistently demonstrates lower read and write times across different collections, indicating its superior schema design compared to the existing models.

### 4.2.6 Scalability

To evaluate scalability, it is essential to test the model's ability to handle increased data volumes while maintaining acceptable query performance. Expanding the data volume in each collection allows us to simulate real-world scenarios with larger datasets and observe how the model performs under such conditions. We can determine if the model scales well with increased data volume by analyzing query performance metrics, such as response times and speed up. This information is crucial for capacity planning and optimizing the database infrastructure to ensure it can handle growing workloads without sacrificing performance. To analyze the scalability, we have increased the data volume, as shown in Table 16, which is almost double the size compared to Table 14. Then, we run the fifteen workload queries (Tables 2 and 9) on CPU and GPU to compare the query performance of the proposed model to the existing models. GPU is chosen due to its ability to accelerate computations, making them vital components of supercomputers.

**4.2.6.1 Scalability for CPU** The results for CPU are shown in Table 17, and the graphical representation is shown in Fig. 20. Based on the following inferences can be drawn:

1. For write queries (Q1, Q2, Q3, Q4, Q14, Q15), the Proposed model consistently outperforms the other models (conventional, optimized, and QPG) with the lowest response times. It demonstrates significant improvements in query performance, indicating better optimization and efficiency in handling write operations.
2. Regarding read queries Q5, Q8, Q12, and Q13, the Proposed and Optimized models perform better than the other two. For queries Q6, Q7, and Q9, the conventional QPG and proposed model achieve lower response times than the optimized model. For Q10 and Q11, the proposed model shows poor query performance than the conventional and optimized but is better than the QPG model. Therefore, the proposed model performs better for seven out of ten read queries (Q5, Q6, Q7, Q8, Q9, Q12, Q13) than the existing models, indicating improved query optimization and data retrieval strategies.
3. The proposed model also outperforms the existing models regarding speedup (SU) for the increased data volume. Specifically, the proposed model performs best against Conventional Model, with SUs of 1.3. It also outperforms Optimized and QPG, with SUs of 1.2.

With increased data volume, the Proposed model consistently exhibits the best performance across both read and write queries, achieving the lowest response times and high speedup factor compared to the Conventional, Optimized, and QPG models. Therefore, with increased data volume, the Proposed model maintains efficient query performance, indicating its ability to handle larger datasets.

**4.2.6.2 Scalability for GPU** To conduct the experimental analysis, we have used an Amazon EC2 P2.xlarge instance with 1 NVIDIA K80 GPU, four vCPU's and 63 GB of RAM. Amazon EC2 is highly optimized for high-performance computing and gives parallel processing capabilities with similar software configurations, as shown in Table 7. The results for GPU are shown in Table 18, and the graphical representation is shown in Fig. 21. Based on the following inferences can be drawn:

1. For write queries (Q1, Q2, Q3, Q4, Q14, Q15), the Proposed model consistently outperforms the other models (conventional, optimized, and QPG) with the lowest response times. It demonstrates significant improvements in query performance, indicating better optimization and efficiency in handling write operations.
2. Regarding read queries Q5, Q8, Q12, and Q13, the Proposed and Optimized models perform better than the other two. For queries Q6, Q7, and Q9, the conventional, QPG, and proposed models achieve lower response times than the optimized model. For Q10 and Q11, the proposed model shows poor query performance than the conventional and optimized but is better than the QPG model. Therefore, the proposed model performs better for seven out of ten read queries

**Table 16** Scaling up the data volume for performance analysis

| Collection name | Conventional | | Optimized | | QPG | | Proposed | |
|---|---|---|---|---|---|---|---|---|
| | Total documents | Disk space (MB) | Total documents | Disk space (MB) | Total documents | Disk space (MB) | Total documents | Disk space (MB) |
| Carrier | 210,000 | 77.44 | 210,000 | 77.44 | 210,000 | 77.44 | 210,000 | 77.44 |
| Category | 27,000 | 2.15 | 27,000 | 2.15 | 27,000 | 2.15 | 27,000 | 2.15 |
| Customer | 520,000 | 1015.84 | 490,000 | 1005.84 | 460,000 | 905.55 | 4900 | 58.16 |
| Product | 255,000 | 29.04 | 285,000 | 39.56 | 90,119 | 7.91 | 285,000 | 39.56 |
| Supplier | 230,000 | 88.36 | 230,000 | 88.36 | 230,000 | 88.36 | 230,000 | 88.36 |
| Payment | 406,000 | 26.6 | – | – | – | – | – | – |
| Order | – | – | – | – | 1,900,000 | 946.08 | 1,900,000 | 946.08 |
| Total storage (MB) | 1239.43 | | 1213.35 | | 2027.49 | | 1211.75 | |

**Table 17** Impact of increased data volume on query response time and query speedup factor

| Q. No | Query operation | Conventional | Optimized | QPG | Proposed | C/P | O/P | QPG/P |
|---|---|---|---|---|---|---|---|---|
| Q1 | Write | 810.0 | 782.0 | 654.0 | 510.0 | 1.6 | 1.5 | 1.3 |
| Q2 | Write | 2505.0 | 2495.5 | 2282.3 | 1996.0 | 1.3 | 1.3 | 1.1 |
| Q3 | Write | 891.0 | 971.2 | 1161.8 | 705.7 | 1.3 | 1.4 | 1.6 |
| Q4 | Write | 18,204.2 | 18,897.0 | 18,872.5 | 13,789.0 | 1.3 | 1.4 | 1.4 |
| Q5 | Read | 21,076.0 | 17,601.8 | 20,477.0 | 12,225.0 | 1.7 | 1.4 | 1.7 |
| Q6 | Read | 22,510.0 | 25,471.0 | 20,317.2 | 19,040.0 | 1.2 | 1.3 | 1.1 |
| Q7 | Read | 13,014.0 | 14,680.0 | 10,909.3 | 9963.0 | 1.3 | 1.5 | 1.1 |
| Q8 | Read | 2067.0 | 1945.0 | 2142.0 | 1511.3 | 1.4 | 1.3 | 1.4 |
| Q9 | Read | 1689.0 | 1769.0 | 1529.3 | 1650.8 | 1.0 | 1.1 | 0.9 |
| Q10 | Read | 5684.0 | 6756.0 | 13,425.0 | 8976.9 | 0.6 | 0.8 | 1.5 |
| Q11 | Read | 2804.0 | 1743.0 | 1875.3 | 2204.0 | 1.3 | 0.8 | 0.9 |
| Q12 | Read | 7661.0 | 4022.0 | 3729.5 | 3006.0 | 2.5 | 1.3 | 1.2 |
| Q13 | Read | 7245.0 | 6646.0 | 5374.0 | 5402.0 | 1.3 | 1.2 | 1.0 |
| Q14 | Write | 1122.5 | 766.0 | 991.5 | 857.0 | 1.3 | 0.9 | 1.2 |
| Q15 | Write | 698.0 | 704.0 | 895.0 | 776.0 | 0.9 | 0.9 | 1.2 |
| Speedup factor | | | | | | 1.3 | 1.2 | 1.2 |

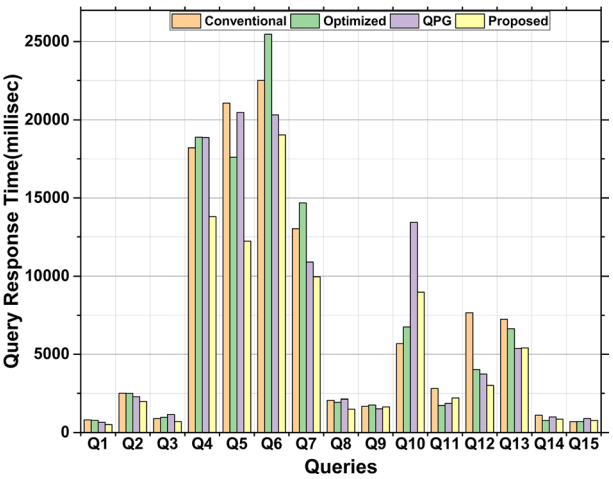


**Fig. 20** Impact of increased data volume on query response time for CPU

(Q5, Q6, Q7, Q8, Q9, Q12, Q13) than the existing models, indicating improved query optimization and data retrieval strategies.

3. The proposed model also outperforms the existing models regarding speedup (SU) for the increased data volume. Specifically, the proposed model performs best against Conventional Model, with SUs of 1.3. It also outperforms Optimized and QPG, with SUs of 1.2.

**Table 18** Query response time and query speedup factor for GPU

| Q. No | Query operation | Conventional | Optimized | QPG | Proposed | C/P | O/P | QPG/P |
|---|---|---|---|---|---|---|---|---|
| Q1 | Write | 67 | 57 | 45 | 38 | 1.8 | 1.5 | 1.2 |
| Q2 | Write | 364 | 356 | 256 | 250 | 1.5 | 1.4 | 1.0 |
| Q3 | Write | 78 | 87 | 95 | 60 | 1.3 | 1.5 | 1.6 |
| Q4 | Write | 2435 | 2768 | 3012 | 2045 | 1.2 | 1.4 | 1.5 |
| Q5 | Read | 5978 | 3856 | 4536 | 3746 | 1.6 | 0.7 | 1.2 |
| Q6 | Read | 10,930 | 13,426 | 10,453 | 8769 | 1.2 | 0.4 | 1.2 |
| Q7 | Read | 5557 | 4964 | 1909 | 3284 | 1.7 | 4.1 | 0.6 |
| Q8 | Read | 386 | 332 | 450 | 225 | 1.7 | 22.1 | 2.0 |
| Q9 | Read | 162 | 199 | 156 | 250 | 0.6 | 0.8 | 0.6 |
| Q10 | Read | 1050 | 2103 | 3425 | 3323 | 0.3 | 0.6 | 1.0 |
| Q11 | Read | 408 | 460 | 389 | 378 | 1.1 | 1.2 | 1.0 |
| Q12 | Read | 1986 | 1356 | 1789 | 1234 | 1.6 | 1.1 | 1.4 |
| Q13 | Read | 1586 | 1384 | 1374 | 1774 | 0.9 | 0.8 | 0.8 |
| Q14 | Write | 150 | 104 | 91.5 | 90 | 1.7 | 1.2 | 1.0 |
| Q15 | Write | 101 | 160 | 234 | 96 | 1.1 | 1.7 | 2.4 |
| Speedup factor | | | | | | 1.2 | 1.4 | 1.2 |

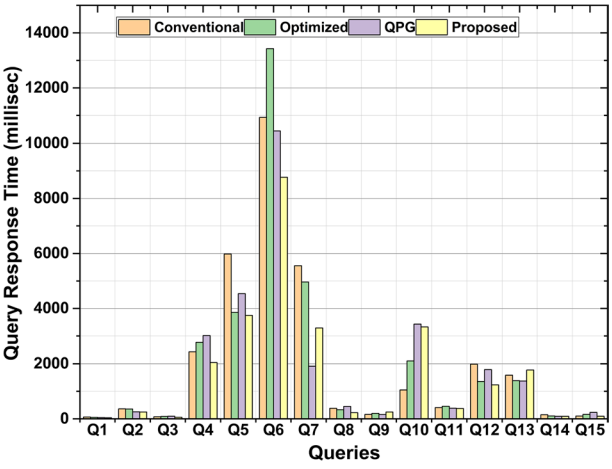


**Fig. 21** Query response time for GPU

As the data volume on the GPU increases, the Proposed model consistently demonstrates superior performance in both read and write queries. It consistently achieves the lowest response times and a substantial speedup factor compared to the Conventional, Optimized, and QPG models.

**4.2.6.3 Throughput and Latency through sharding** MongoDB achieves scalability by utilizing a horizontal scaling technique named sharding because they are specifically designed to provide horizontal scalability to meet modern applications' high data volume demands. Sharding [7] is used to partition data horizontally across multiple servers or shards. It allows distributing the data across the cluster of machines to increase data storage capacity and improve query performance. To check the scalability of the proposed model, we have experimented by distributing the data among different clusters (nodes), as shown in Table 19. Our proposed model is deployed on MongoDB Atlas, which provides sharding capabilities to efficiently distribute the data. The general process for distributing data among different clusters in MongoDB:

(a) Set up the Clusters: Create individual clusters to host the shards. Each cluster should have its own set of servers running MongoDB instances. To experiment, we have created 2, 3, and 4 Clusters having 3 nodes each.
(b) Enable Sharding: Enable sharding on the clusters by configuring the config servers and enabling sharding for the relevant databases or collections.
(c) Define Sharding Key: The sharding key determines how data is divided across the clusters. There are various sharding keys named range-based, hash-based, and compound-based. We have chosen hash-based keys for our work because it automatically distributes the documents uniformly across the shards.
(d) Distribute Data: Insert or migrate data into the sharded collections. MongoDB distributes the data across the shards based on the defined sharding key. The sharded clusters ensure load balancing of read and write operations and uniform data distribution.

After establishing the experimental setup, we proceeded with conducting experiments to evaluate the scalability of the proposed model using sharding on the scaled data, as detailed in Table 16. Specifically, we focus on two key parameters across the distributed data on different nodes: (a) Throughput (Sect. 4.2.7.1), and (b) Latency (Sect. 4.2.7.2).

**Table 19** Details of sharded clusters

| Parameters | 3 nodes | 6 nodes | 9 nodes | 12 nodes |
|---|---|---|---|---|
| No. of Sharded Clusters | 1 | 2 | 3 | 4 |
| Replication | 3 | 3 | 3 | 3 |
| Primary Node | 1 | 2 (1 per shard) | 3 (1 per shard) | 4 (1 per shard) |
| Secondary Nodes | 2 | 4 (2 per shard) | 6 (2 per shard) | 8 (2 per shard) |
| No. of Config servers | 1 | 2 | 3 | 4 |
| Shard Key | Hash-based | Hash-based | Hash-based | Hash-based |
| Distributed data | Balanced | Balanced | Balanced | Balanced |

**4.2.6.4 Throughput** Throughput refers to how much work or data can be processed by the database system within a given time frame. It represents the rate at which MongoDB can handle and process operations, such as reads, writes, and queries. To maximize throughput in MongoDB, it is recommended to carefully design the database schema, optimize queries, utilize appropriate indexes, and scale the deployment horizontally by adding more servers or shards as needed. To calculate the throughput of the distributed data on different nodes, Apache JMeter—a tool known for measuring metrics like throughput and latency. Various test cases for CRUD operations are designed to compare the performance of proposed models against existing ones. Table 20 provides the measured throughput (operations per minute(ops/mint)) for different models on varying numbers of nodes. The graphical representation is shown in Fig. 22. For the Conventional model, the throughput ranges from 510 ops/mint (3 nodes) to 1003 ops/mint (12 nodes). There is slightly better throughput for the Optimized model than the Conventional model, ranging from 514 ops/mint (3 nodes) to 1128 ops/mint (12 nodes). For the QPG model, the throughput ranges from 501 ops/mint (3 nodes) to 1045 ops/mint (12 nodes. For the Proposed model, the throughput is highest, ranging from 518 ops/mint (3 nodes) to 1169 ops/mint (12 nodes). Overall, the results indicate that the proposed model achieves the highest throughput, followed by the optimized and QPG models. Furthermore, as the number of nodes increases, there is a general trend of improved throughput across all models, demonstrating the benefits of horizontal scaling.

**4.2.6.5 Latency** Latency refers to the delay between requesting and receiving a response from the server. While sharding can improve scalability and throughput, it can introduce additional latency due to the system's distributed nature. The latency (milliseconds(ms)) results for data distribution over different nodes are presented in Table 21. Figure 23 provides a graphical representation of the latency. The conventional model demonstrates low latency, ranging from 3.9 ms (3 nodes) to 27 ms (12 nodes). The optimized model has constant performance across node configurations, with latency ranging from 3.5 ms (3 nodes) to 29.7 ms (12 nodes). The latency of the QPG model ranges from 4.1 ms (3 nodes) to 29.1 ms (12 nodes), which is equivalent to the conventional and optimized models. The proposed model has constant latency ranging from 3.3 ms (3 nodes) to 28.4 ms (12 nodes), similar to the other models. Overall, there is a modest rising trend in latency as the number of nodes grows, indicating that processing time may increase. In conclusion, all four models demonstrate relatively low and comparable latency, with different node variations.

**Table 20** Throughput (ops/mint) comparison on various distributed nodes

| Nodes | Conventional | Optimized | QPG | Proposed |
|---|---|---|---|---|
| 3 | 510 | 514 | 501 | 518 |
| 6 | 619 | 635 | 605 | 698 |
| 9 | 856 | 901 | 845 | 920 |
| 12 | 1003 | 1128 | 1045 | 1169 |

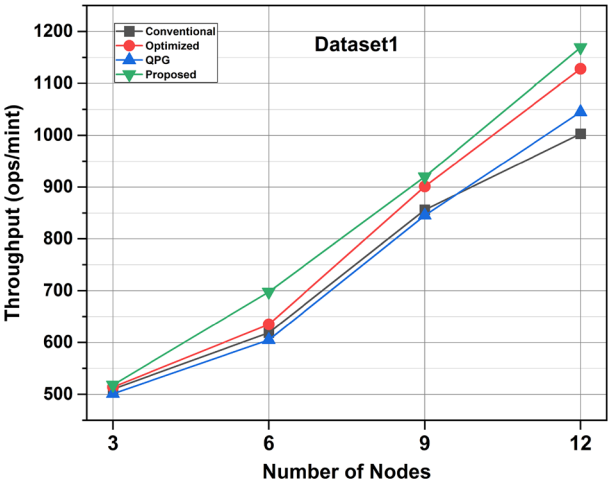


**Fig. 22** Throughput (ops/mint) on various distributed nodes

## 5 Conclusion

Designing a NoSQL database schema requires not only knowledge of data but also an understanding of how the application needs to access the data. This paper presents an automatic workload-driven model for the logical schema of a document-based NoSQL database from a conceptual model. The model takes the conceptual model and the application workload in estimated data volume and query workload. The query graphs are generated from the application workload to study the query characteristics. The characteristics are represented using query labels. These labels are used to transform the conceptual model into MongoDB logical schema.

This paper has designed a model to minimize data modeling hardships for the popular database named MongoDB. The proposed model does not rely on rules of thumb to select the appropriate schema or require expert help to design a logical schema. Therefore, the proposed work benefits novice programmers and helps them to save time for schema design decisions during the early development phase of any application's design.

We employed three state-of-the-art schema generation models termed conventional, optimized, and QPG to validate the performance of the proposed model. Several parameters, including query response time, query speedup factor, read and write

**Table 21** Latency on various distributed nodes

| Nodes | Conventional | Optimized | QPG | Proposed |
|---|---|---|---|---|
| 3 | 3.9 | 3.5 | 4.1 | 3.3 |
| 6 | 14 | 12.5 | 15.2 | 13.8 |
| 9 | 21 | 20.5 | 22.2 | 20.8 |
| 12 | 27 | 29.7 | 29.1 | 28.4 |

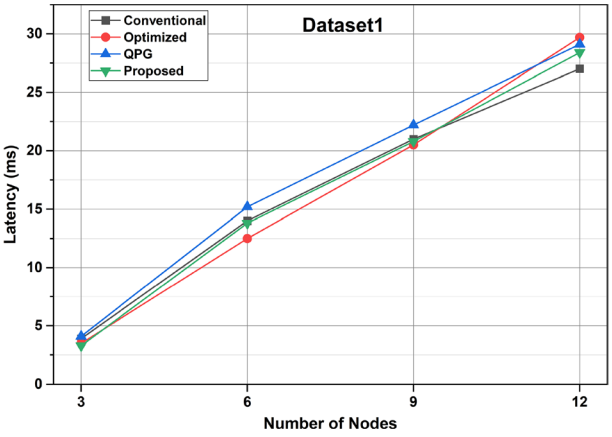


**Fig. 23** Latency comparison on distributed nodes

latency, aggregate pipeline efficiency improvement, storage space, collection-wise performance, and scalability, are used to compare the proposed model to existing models: Conventional, Optimized, and QPG. The experimental results show the proposed model outperforms conventional, optimized, and QPG models. It achieved a 1.2, 1.1, and 1.3 speedup factor over the respective models. Additionally, the proposed model improved aggregate pipeline efficiency by 17.5%, 15%, and 10% compared to the conventional, optimized, and QPG models. The proposed model showcased advantages in terms of good performance in terms of storage space utilization, with lower read and write latencies. It exhibited good performance when scaling the volume to double. Furthermore, the proposed model enables the system to efficiently handle growing data volumes by implementing horizontal scaling techniques, resulting in high throughput and low latency. This highlights the efficiency and effectiveness of our model in handling distributed data scenarios. Based on these findings, it is evident that the proposed model surpasses the existing models (Conventional, Optimized, and QPG) in multiple aspects, including query performance, storage space efficiency, aggregate pipeline efficiency, read–write latency, collection-wise performance, scalability, throughput and latency. Therefore, the proposed model effectively tackles the challenges associated with managing the variety and volume of big data through the well-designed schema. This schema design significantly improves system performance and guarantees scalability for datasets of any size. As future work we intend to expand the similar concept to other NoSQL categories, like column and graph databases.

**Acknowledgements** The authors would like to thank the editor and anonymous reviewers whose insightful comments helped to improve the readability and quality of this paper.

**Author contributions** NB: Writing Original Draft, Writing—Reviewing and Editing, Conceptualization, Methodology, Programming, Validation, SS: Supervision, Validation, Writing—Reviewing and Editing, LKA: Supervision, Validation, Writing—Reviewing and Editing.

**Funding** Not Applicable.

**Data availability** Available on Request.

## Declarations

**Conflict of interest** Not Applicable.

## References


1. Davoudian A, Chen L, Liu M (2018) A survey on NoSQL stores. ACM Comput Surv. https://doi.org/10.1145/3158661
2. Patel JM (2016) Operational NoSQL systems: What's new and what's next? Computer 49:23–30. https://doi.org/10.1109/MC.2016.118
3. Azad P, Navimipour NJ et al (2020) The role of structured and unstructured data managing mechanisms in the Internet of things. Cluster Comput. https://doi.org/10.1007/s10586-019-02986-2
4. Faccia A, Cavaliere LPL, Petratos P, Mosteanu NR (2022) Unstructured over structured, big data analytics and applications in accounting and management. In: Proceedings of the 2022 6th International Conference on Cloud and Big Data Computing, pp 37–41. https://doi.org/10.1145/3555962.3555969
5. Stonebraker M (2010) SQL databases v. NoSQL databases. Commun ACM 53:10–11. https://doi.org/10.1145/1721654.1721659
6. Vera-Olivera H, Guo R, Huacarpuma RC et al (2021) Data modeling and NoSQL databases-a systematic mapping review. ACM Comput Surv. https://doi.org/10.1145/3457608
7. Database Sharding: Concepts and Examples. https://www.mongodb.com/features/database-sharding-explained
8. Wang L, Zhang S, Shi J et al (2015) Schema management for document stores. Proc VLDB Endow 8(9):922–933. https://doi.org/10.14778/2777598.2777601
9. Gómez P, Roncancio C, Casallas R (2021) Analysis and evaluation of document-oriented structures. Data Knowl Eng 134:101893. https://doi.org/10.1016/j.datak.2021.101893
10. Mior MJ, Salem K, Aboulnaga A, Liu R (2017) NoSE: schema design for NoSQL applications. IEEE Trans Knowl Data Eng 29:2275–2289. https://doi.org/10.1109/TKDE.2017.2722412
11. Li C (2010) Transforming relational database into HBase: a case study. In: Proceedings 2010 IEEE International Conference on Software Engineering and Service Sciences, ICSESS 2010, pp 683–687. https://doi.org/10.1109/ICSESS.2010.5552465
12. Ceresnak R, Dudas A, Matiasko K, Kvet M (2021) Mapping rules for schema transformation : SQL to NoSQL and back. In: International Conference on Information and Digital Technologies 2021, IDT 2021, pp 52–58. https://doi.org/10.1109/IDT52577.2021.9497629
13. Imam AA, Basri S, Ahmad R et al (2018) Data modeling guidelines for NoSQL document-store databases. Int J Adv Comput Sci Appl 9:544–555. https://doi.org/10.14569/IJACSA.2018.091066
14. De Lima C, Dos Santos Mello R (2015) A workload-driven logical design approach for NoSQL document databases. In: 17th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2015 - Proceedings. https://doi.org/10.1145/2837185.2837218
15. Jia T, Zhao X, Wang Z, D Gong (2016) Model transformation and data migration from relational database to MongoDB. In: 2016 IEEE International Congress on Big Data (BigData Congress)
16. Kuszera EM, Peres LM, Didonet Del Fabro M (2022) Exploring data structure alternatives in the RDB to NoSQL document store conversion process. Inf Syst 105:101941. https://doi.org/10.1016/j.is.2021.101941
17. Chen L, Davoudian A, Liu M (2022) A workload-driven method for designing aggregate-oriented NoSQL databases. Data Knowl Eng 142:102089. https://doi.org/10.1016/j.datak.2022.102089
18. DB-Engines Ranking - popularity ranking of relational DBMS. https://db-engines.com/en/ranking/relational+dbms. Accessed 21 Jun 2022
19. Rodríguez-Mazahua L, Rodríguez-Enríquez CA, Sánchez-Cervantes JL et al (2016) A general perspective of big data: applications, tools, challenges and trends. J Supercomput 72:3073–3113. https://doi.org/10.1007/s11227-015-1501-1

20. Rabl T, Sadoghi M, Jacobsen HA et al (2012) Solving big data challenges for enterprise application performance management. Proc VLDB Endow 5:1724–1735. https://doi.org/10.14778/2367502.2367512
21. da Silva LF, Lima JVF (2023) An evaluation of relational and NoSQL distributed databases on a low-power cluster. J Supercomput. https://doi.org/10.1007/s11227-023-05166-7
22. Ko HKE, Lee YJK (2020) Techniques and guidelines for effective migration from RDBMS to NoSQL. J Supercomput 76:7936–7950. https://doi.org/10.1007/s11227-018-2361-2
23. Khatibi E, Mirtaheri SL (2019) A dynamic data dissemination mechanism for cassandra NoSQL data store. J Supercomput 75:7479–7496. https://doi.org/10.1007/s11227-019-02959-7
24. Zilio D, Rao J, Lightstone S, et al. (2004) DB2 Design advisorintegrated automatic physical database design. In: Proceedings 2004 VLDB Conference, pp 1087–1097. https://doi.org/10.1016/b978-012088469-8/50095-4
25. Bruno N, Chaudhuri S (2005) Automatic physical database tuning: a relaxation-based approach. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 227–238. https://doi.org/10.1145/1066157.1066184
26. Roy-Hubara N, Sturm A (2020) Design methods for the new database era: a systematic literature review. Softw Syst Model 19:297–312. https://doi.org/10.1007/S10270-019-00739-8/TABLES/1
27. Roy-Hubara N (2019) The quest for a database selection and design method. CEUR Workshop Proc 2370:69–77
28. Störl U, Klettke M, Scherzinger S (2020) NoSQL schema evolution and data migration: State-of-the-art and opportunities. Adv Database Technol. https://doi.org/10.5441/002/edbt.2020.87
29. Gómez P, Casallas R, Roncancio C (2016) Data schema does matter, even in NoSQL systems!. In: Proceedings - International Conference on Research Challenges in Information Science 2016-Augus:1–6. https://doi.org/10.1109/RCIS.2016.7549340
30. Mior MJ (2014) Automated schema design for NoSQL databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 41–45. https://doi.org/10.1145/2602622.2602624
31. Hewasinghage M, Nadal S, Abelló A, Zimányi E (2023) Automated database design for document stores with multicriteria optimization. Knowl Inf Syst 65:3045–3078. https://doi.org/10.1007/s10115-023-01828-3
32. Roy-Hubara N, Sturm A, Shoval P (2023) Designing NoSQL databases based on multiple requirement views. Data Knowl Eng 145:102149. https://doi.org/10.1016/j.datak.2023.102149
33. Imam AA, Basri S, Ahmad R, González-Aparicio MT (2019) Schema proposition model for NoSQL applications. Adv Intell Syst Comput 843:30–39. https://doi.org/10.1007/978-3-319-99007-1_3
34. Imam AA, Basri S, Ahmad R et al (2020) Dsp: schema design for non-relational applications. Symmetry 12:1–33. https://doi.org/10.3390/sym12111799
35. Chebotko A, Kashlev A, Lu S (2015) A big data modeling methodology for apache cassandra. In: Proceedings of the 2015 IEEE International Congress on Big Data, Bigdata Congress 2015, pp 238–245. https://doi.org/10.1109/BigDataCongress.2015.41
36. Jia T, Zhao X, Wang DG-2016 II, 2016 U (2016) Model transformation and data migration from relational database to MongoDB. In: In 2016 IEEE International Congress on Big Data (BigData Congress), pp 60–67
37. Lima C, Mello RS (2016) On proposing and evaluating a NoSQL document database logical approach. Int J Web Inf Syst 12:398–417. https://doi.org/10.1108/IJWIS-04-2016-0018
38. Reniers V, Van Landuyt D, Rafique A, Joosen W (2017) Schema design support for semi-structured data: Finding the sweet spot between NF and De-NF. In: Proceedings of the 2017 IEEE International Conference on Big Data, Big Data 2017 2018-Jan, pp 2921–2930. https://doi.org/10.1109/BigData.2017.8258261
39. Davoudian A (2021) A workload-driven framework for NoSQL data modeling and partitioning, PhD Dissertation. Carleton University
40. Hewasinghage M, Abelló A, Varga J, Zimányi E (2021) A cost model for random access queries in document stores. VLDB J 30:559–578. https://doi.org/10.1007/s00778-021-00660-x
41. Hewasinghage M, Abelló A, Varga J, Zimányi E (2020) DocDesign: cost-based database design for document stores. In: 32nd International Conference on Scientific and Statistical Database Management (SSDBM), ACM, pp 1–4. https://doi.org/10.1145/3400903.3401689
42. Engels G, Gogolla M, Hohenstein U et al (1992) Conceptual modelling of database applications using an extended ER model. Data Knowl Eng 9:157–204. https://doi.org/10.1016/0169-023X(92)90008-Y

43. Pirahesh H, Hellerstein JM, Hasan W (1992) Extensible/rule based query rewrite optimization in starburst. ACM SIGMOD Rec 21:39–48. https://doi.org/10.1145/141484.130294
44. Rosenthal A, Galindo-Legaria C (1990) Query graphs, implementing trees, and freely-reorderable outerjoins. Proc ACM SIGMOD Int Conf Manage Data 1990:291–299
45. Data Modeling Introduction — MongoDB Manual. https://www.mongodb.com/docs/upcoming/core/data-modeling-introduction/. Accessed 26 Jun 2022
46. What Customer Lifetime Value (CLV) Is & How to Calculate It | NetSuite. https://www.netsuite.com/portal/resource/articles/ecommerce/customer-lifetime-value-clv.shtml. Accessed 19 Jan 2023
47. The Professional Client, IDE and GUI for MongoDB | Studio 3T. https://studio3t.com/. Accessed 8 Jun 2023
48. Fleming PJ, Wallace JJ (1986) How not to lie with statistics: the correct way to summarize benchmark results. Commun ACM 29:218–221. https://doi.org/10.1145/5666.5673
49. Henry OB (2019) MongoDB aggregation stages and pipelining. White paper, pp 1–38



## Authors and Affiliations

**Neha Bansal[1] · Shelly Sachdeva[1] · Lalit K. Awasthi[2]**

✉ Shelly Sachdeva
shellysachdeva@nitdelhi.ac.in

Neha Bansal
nehagoel@nitdelhi.ac.in

Lalit K. Awasthi
lalit@nith.ac.in

[1] National Institute of Technology Delhi, Delhi, India

[2] National Institute of Technology Hamirpur, Hamirpur, India