American Journal of Computer Science and Technology
2025, Vol. 8, No. 2, pp. 22-41
https://doi.org/10.11648/j.ajcst.20250802.12

Research Article

Multi-model Database Design and Query Processing in NewSQL DBMSs

Joachim Tankoano*

Institut Burkinabè des Arts et des Métiers, Université Joseph Ki-Zerbo, Ouagadougou, Burkina Faso

Abstract

NewSQL DBMSs are hybrid systems that combine the advantages of both SQL DBMSs and NoSQL DBMSs. This paper proposes a method for designing a database to be implemented on a NewSQL DBMS. The objectives of this method are identifying and defining heterogeneous collections of values that adhere to different data models, which are essential for an enterprise's operations, with the goal of storing and managing them within only one database. This method is based on a Relational Data Store and the Nested Relational Model. It allows the designer to use the Data Store as a guarantor of integrity and to optimize it for hybrid workloads (transactional and analytical), performance, scalability, and continuous data availability. As for the nested relational model, it is used by the designer to: (1) clarify their choices regarding storage models that can enable fast access to data about complex real-world entities; (2) specify access paths that can meet user needs. The main interest and the originality of this methodological approach are that this enables us to use the Nested Relational Model as a Pivot Model to: (1) automatically generate the global external schemas of the NoSQL virtual databases, allowing users to view and manipulate the Data Store as if it were a NoSQL database (object-relational, XML, JSON, or graph-oriented), and (2) unify the processing of cross-model SQL queries through an innovative and efficient approach. This method consistently integrates, through five levels of abstraction, the design process of the relational Data Store and that of the virtual databases. The research method used consisted of: (1) defining the objectives of this approach, (2) identifying the required levels of abstraction in light of the targeted objectives, (3) determining, for each level of abstraction, its specific objectives as well as the role to be played by the designer, a design support tool, and the DBMS, and (4) applying this approach to a typical example reflecting the most common needs, in order to facilitate the understanding of its contributions and relevance with respect to the intended objectives.

Keywords

Multi-model Data, NewSQL, NoSQL, Database Design, Query Processing

1. Introduction

NewSQL DBMSs are hybrid systems [1-13]. Their advent aims to promote new types of DBMSs with the required capability to better address the requirements of web applications compared to SQL DBMSs and NoSQL DBMSs. To achieve this, they focus on two key objectives:

Maintain the advantages of SQL DBMSs by offering: (i) capabilities for managing fixed-schema collections of values that ensure data independence and integrity; (ii) a non-procedural interface language that facilitates query optimization; (iii) a transaction execution environment that

*Corresponding author: tankoanoj@gmail.com (Joachim Tankoano)

Received: 4 April 2025; **Accepted:** 17 April 2025; **Published:** 9 May 2025

guarantees the Atomicity, Consistency, Isolation, and Durability (ACID) properties.

Consider the advantages of NoSQL DBMSs, which include: (i) flexibility through flexible-schema or schema-less collections of values, adhering to various structures (e.g. structured, hierarchical-and-structured, hierarchical-and-semi-structured, unstructured, or graph-oriented); (ii) customizable data logical and physical organization enabling optimal query execution; (iii) unlimited horizontal scaling to accommodate workload changes; (iv) continuous data availability.

Many of today's NewSQL DBMSs have been developed by leveraging a proven SQL DBMS and incorporating well-known concepts to provide the necessary functionality. This include:

The integration of alternative NoSQL data models (c.f. the SQL3, SQL/XML, SQL/JSON and SQL/PGQ ISO standards) achieved by: (i) extending SQL as a complete programming language or integrating SQL into an existing one, that can in both cases serve as support for integration of alternative data models under the relational model, and for development of features and APIs that extend the capabilities of the DBMS; (ii) expanding the SQL data definition language with new value types, structure types, and table types; (iii) expanding SQL's relational predicative language with new functions and predicates; (iv) adding new operators and clauses to SQL syntax; (v) defining the semantics of the operators of nested relational algebra as an extension of 1NF relational algebra; (vi) using these operators to define the semantics of path expressions in object-oriented/document-oriented models; (vii) allowing nested queries in SQL SELECT-FROM-WHERE clauses; (viii) defining a cross-model query language as an extension of SQL 2.

The integration of a middleware into DBMS architecture for automatic and transparent sharding of the database, including: (i) automatically fragmenting horizontally tables and indexes, and distributing fragments and their copies across geographically distributed database servers called shards; (ii) monitoring shards, rebalancing them, and adjusting their number based on workload; (iii) routing queries and coordinating concurrent, distributed transactions while ensuring Atomicity, Consistency, Isolation, and Durability (ACID) properties.

The incorporation of functionalities that enable DBMSs to offer both transactional and analytical processing capabilities.

Following the approach described, NewSQL DBMS have introduced a broad range of possibilities, thereby increasing the complexity of database design.

Furthermore, some of these possibilities open the door to very poor practices.

To our knowledge, there is currently no research on comprehensive methods for designing a database specifically aimed at NewSQL DBMSs, with the following objectives: (1) identifying and defining heterogeneous collections of values, adhering to different data models, for storage and manipulation within only one database, ensuring they meet the needs of a business's operations; (2) guaranteeing performance, scaling capability, and continuous data availability.

This paper outlines such a coherent and comprehensive methodological approach for designing a database specifically intended for a NewSQL DBMS.

This approach is grounded on the ANSI/SPARC architecture of the schemas depicted in Figure 1 as described in [14]. The key point of this architecture is that the logical model of the database utilized internally as the Data Store must conform to the relational model.

This method allows the designer to use this relational Data Store as the guarantor of integrity and to optimize it for hybrid processing workloads (both transactional and analytical), performance, scaling, and continuous data availability.

Furthermore, this method allow the designer to leverage the Nested Relational Model to: (1) clarify their choices regarding storage models that can enable fast access to data about complex real-world entities; (2) specify access paths that can meet user needs.

In doing so, this method enables us to use the Nested Relational Model as a Pivot Model to: (1) automatically generate the global external schemas of the NoSQL virtual databases, allowing users to view and manipulate the Data Store as if it were a NoSQL database (object-relational, XML, JSON, or graph-oriented), and (2) unify the processing of cross-model SQL queries through an innovative and efficient approach.

This method integrates seamlessly the design methodologies of various types of databases (relational and NoSQL).

It captures the various relevant aspects of a multi-model database across five levels of abstraction, which are complementary but also consistent or orthogonal.

As depicted in Figure 2, these five levels concern two design processes: (1) the relational Data Store design process; (2) the design process for the virtual NoSQL databases.

The following presents and illustrates the capture of the different relevant aspects of a multi-model database through these five levels of abstraction by relying on a simple example of database and highlighting the known techniques that can be considered.

A simple example of generation and optimization of the logical execution plan of a query formulated on this example of a multi-model database is also presented.

The paper ends with a comparative analysis of the presented approach and related works, followed by a conclusion.

2. The Database Conceptual Schema

The conceptual schema of a multi-model database within a NewSQL DBMS corresponds to the conceptual representation used in our approach to identify: (i) real-world entities and their associations whose instances need to be described in this database, independently of use cases such as the models that must be used for the description and manipulation of value collections, as well as other non-functional requirements that

must be met (e.g., value access path, value access time, security for access, scaling capability, data availability, etc.); (2) atomic value types required for this description; (3) integrity constraints that must be applied to these values, entities and associations to ensure consistency with the enterprise's business rules.

In our approach, the role of this conceptual schema is also to facilitate derivation of the global logical schema of a multi-model database for storage and manipulation of: (1) the collections of structured values (adhering to the relational model) describing the instances of simple entities; (2) the collections of hierarchical-and-structured values (adhering to the nested relational model) describing the instances of complex entities with nested sub-entities; (3) the collections of hierarchical-and-semi-structured values whose structure is irregular and/or unpredictable (adhering to XML or JSON standards) describing the instances of what we refer to as gray entities (such as the entity "*Article*" in an online store where articles of diverse natures (food, clothing, household appliances, etc.) are described by their suppliers according to different rules).

Accordingly, this conceptual schema must highlight the real-world entities that need to be described in the database by clearly differentiating among three types of entities: simple entities, complex entities, and grey entities.

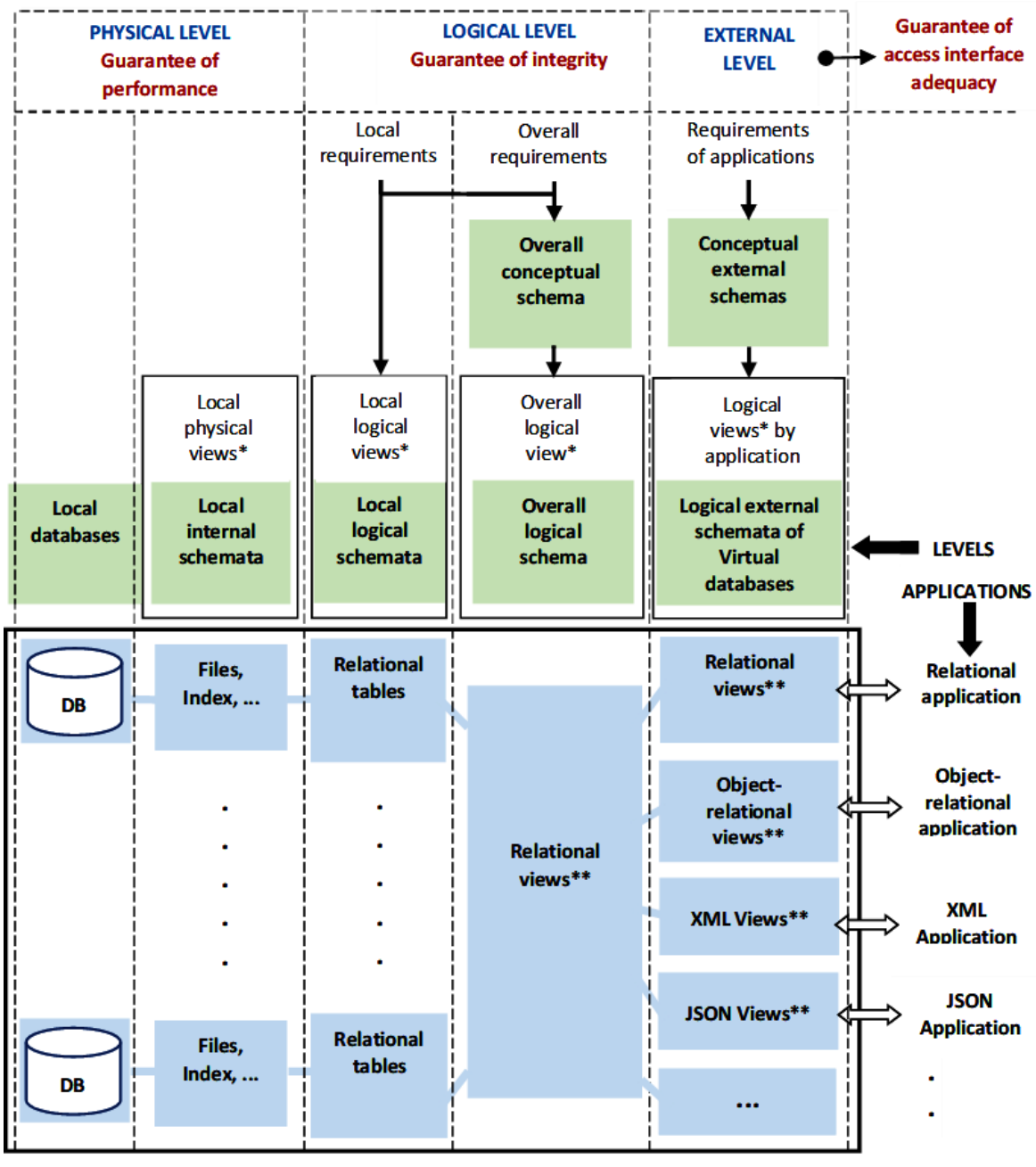

Figure 1. *ANSI/SPARC architecture of schemas [14].*

	Design process of the Relational data store	SCHEMAS corresponding in the methodology to the five levels of abstraction of the database	Design process of the virtual NoSQL databases
Conceptual Level	√	*UML schema* that represents User Information Requirements	√
Logical Level	√	*Global Relational Schema of the Data Store for Structured and Semi-Structured Data* Designed to Prevent Storage Anomalies and Provide the DBMS with the necessary capabilities to Ensure Data Integrity	√
	√	*Global Nested Relational Schema* Designed to further optimize data access time on complex entity instances with regular or irregular structures	√
Physical Level	√	Specifications on how Data is Stored and Optimized for Performance, Scalability, and Data Availability	
External Level		*Global External Schemas*, which Allow the Data Store to be Viewed and Manipulated as if it were NoSQL Virtual Databases, where the Required Access paths have been Considered	√

Figure 2. *The two design processes and the five levels of abstraction of the database.*

Each real-world entity must be characterized within this conceptual schema, according to the users' information requirements, through the atomic-valued attributes of its instances.

For simple entities and complex entities with sub-entities, the type of an attribute corresponding to a large and/or unstructured value (e.g., text, image, audio, video), must be a special atomic abstract type ("*LOB (Large OBject)*", "*Character*", "*Character Varying*" …) which is part of the extensions introduced by ISO from SQL3. This also applies to repetitive values (e.g., a customer's phone numbers) or values with varying structures (e.g., a customer identity which may consist of either his title (Ms, Mrs, Pr, Dr, etc.), his surname and first name, or his first name and surname, or the acronym and name of the enterprise).

In this conceptual schema, complex entities should be distinguished by explicitly identifying in each association which entity is a sub-entity. A sub-entity is the entity whose existence of each of its instances depends on the existence of only one instance of the other entity with which it is it associated.

As for gray entities, they must be differentiated by characterizing them using only two attributes. The first attribute must be used to provide the value that uniquely identifies each instance of the gray entity in question, and the second attribute to provide the hierarchical-and-semi-structured value that describes this instance. The type of this second attribute at the conceptual level must be a special atomic abstract type ("*LOB (Large OBject)*", "*Character*", "*Character Varying*" …). When for each instance of a gray entity the data that describe it is used to supplement the data that describe an instance of another non-gray entity, these two entities must be linked by a "*1:1*" association. In this case, we will say that the gray entity is a gray entity detached from the non-gray entity to which it is associated.

The most common formalisms for specifying this type of representation are the conceptual data models "*E/A (Entity/Association)*" [15] and "*UML (Unified Modeling Language)*" [16].

In this paper, we utilize a UML-based formalism in which: (1) each entity is materialized by a UML class; (2) each sub-entity is distinguished by a dotted class symbol and a dotted association symbol that denotes its dependency; (3) each gray entity is differentiated using a gray background class symbol and possibly a dotted association symbol which indicates that it is a gray entity detached from the non-gray entity with which it is associated.

The UML schema in Figure 3 is the conceptual representation for the example database used in this paper. In this schema, two associations are depicted using a dotted line, indicating that the entities represented by the "*Post*" and "*Composition*" classes are sub-entities of the complex entity "*Composer*". Each instance of these two sub-entities must depend on only one single instance of the complex entity "*Composer*" to which it belongs.

3. Transforming into a Global Relational Logical Schema

In the schema architecture shown in Figure 1, the global relational logical schema is that of the Data Store. Its role is to specify the logical representation that results from the engineering of the schemas of the value collections within the Data Store. The finality of the design of these schemas must be to prevent storage anomalies and to allow DBMS to ensure integrity of these value collections.

In the multi-model database design process presented in this paper, its global relational logical schema, where each relation schema is in 3NF, can be advantageously derived from the conceptual schema. This derivation relies on well-established transformation rules [14], which enable the DBMS to enforce the integrity constraints defined in the conceptual schema by the designer, based on the enterprise's business rules. In the resulting global relational schema, in-

tegrity constraints related to multivalued dependencies that were not explicitly defined in the conceptual schema must be addressed by the designer. This is done by decomposing the affected relations to ensure that each relation schema is in 4NF.

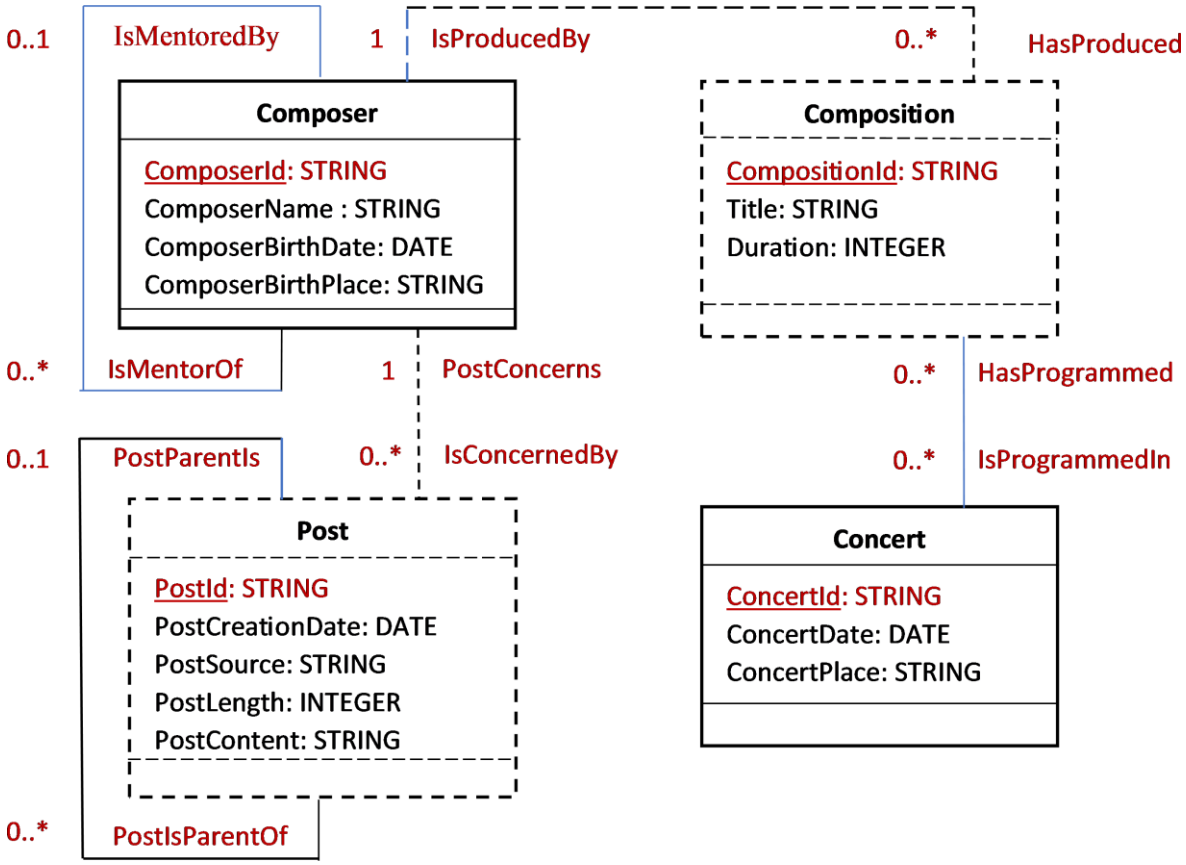

Figure 3. *Example of a UML Conceptual Schema for a Database.*

To accurately model complex entities as described in the conceptual schema, it is necessary to represent data about a sub-entity using a relation that includes a foreign key in its primary key. This foreign key must reference the primary key of the relation corresponding to the entity with which it is directly associated in the conceptual schema. Relations of this type are considered weak relations [15]. This approach materializes data about each complex entity in the global relational schema of the Data Store using a set of relations connected within a tree structure where each child includes a foreign key that references the primary key of its parent.

Likewise, each gray entity in the conceptual schema must be represented in this global relational logical schema by a relation. The designer must specify in this relation the type of NoSQL model (XML or JSON) to which each hierarchical-and-semi-structured value describing an instance of this gray entity must adhere. This includes specifying the type of the attribute that designates this value and, if applicable, associating it with a schema that defines its internal structure. Moreover, if the gray entity is detached, in this relation and in the relation that materializes the non-gray entity from which it was detached, the primary key must be defined as also being a foreign key referencing the primary key of the other relation.

Additionally, for every attribute (of a simple or complex entity) corresponding to repetitive values or values with varying structures, the designer must specify the type of model he wishes to use for this attribute (either XML or JSON). He should also define a schema if the structure of these values can be predetermined.

The content of Figure 4 provides an example of the transformation of the conceptual schema of the database, defined in Figure 3, into a global relational schema. This transformation has been carried out in compliance with the general principles outlined above.

In the resulting global relational schema, data about the instances of the complex entity "*Composer*" and its sub-entities "*Post*" and "*Composition*" are stored in the relations "*Composers_R*", "*Posts_R*", and "*Compositions_R*", respectively. The foreign key "*ComposerId+*" in the "*Posts_R*" relation is part of its primary key. Similarly, the foreign key "*ComposerId+*" in the "*Compositions_R*" relation is also part of its primary key.

Composers_R (ComposerId, ComposerName, ComposerBirthDate, ComposerBirthPlace, IsMentoredBy+)
Posts_R (ComposerId+, PostId, PostCreationDate, PostSource, PostLenght, PostContent, PostParentIs+)
Compositions_R (ComposerId+, CompositionId, Title, Duration)
Concerts_R (ConcertId, ConcertDate, ConcertPlace)
Programs_R (ComposerId CompositionId+, ConcertId+)

Figure 4. *Example of a Global Relational Schema for the Data Store.*

This global relational schema prevents anomalies that may arise during insert, update, and delete operations. Additionally, it enables the DBMS to enforce database integrity, as defined in the conceptual schema regarding the management rules.

4. Transforming into a Global Nested Relational Logical Schema

Storage models are among the main means that a database designer must use depending on the use cases to enhance the speed of data access. For the data about instances of complex entities or detached gray entities, the two most recognized storage models are the direct storage model and the normalized storage model [17, 18].

The direct storage model of the data that describe the instances of a complex entity leads to logically grouping these data into only one table where each row contains the data about a single instance including the data of all the instances of the sub-entities that belong to it, arranging these data in a hierarchical manner regardless of their nesting levels [17]. This storage model is the one that is most suitable when the main need of the applications is to be able to access all or part of the data of each instance concerning this complex entity, whenever necessary, using the shortest possible time.

Let "*Eg*" be a gray entity detached from the non-gray entity "*Eng*". The storage of the data describing the instances of "*Eg*" according to the direct storage model is accomplished by moving the attribute used to store these data from the relation corresponding to "*Eg*" to the relation corresponding to "*Eng*", followed by the removal of the relation corresponding to "*Eg*". This storage model is the most suitable when the primary requirement of the applications is to be able to access at the same time the data of each instance of "*Eng*" and the detached data in "*Eg*" which concern it, whenever necessary, while ensuring the shortest possible access time.

The normalized storage model for the data about instances of a complex entity leads to distributing these data in several tables resulting from a normalization process by decomposition [17]. The purpose of this decomposition may be, for example, to ensure that there is, on the one hand, a dedicated table for storing the data relating to instances of the enclosing entity and, on the other hand, for each sub-entity a dedicated table for storing the data relating to its instances. For this example of decomposition, this storage model is best when the primary requirement for applications is to be able to directly access the data about the instances of each sub-entity, regardless of their nesting levels, without having to go through the data about the instance of the entity to which they belong, using the shortest possible time.

Let "*Eg*" be a gray entity detached from the non-gray entity "*Eng*". The storage of the data describing instances of "*Eg*" according to the normalized storage model is done by representing "*Eg*" and "*Eng*" using separate relations. This storage model is the most suitable when the primary requirement of applications is to be able to manage and access separately the data that describes instances of "*Eg*" and the data that describes instances of "*Eng*".

The direct storage model and the normalized storage model can be combined, to allow storing part of the data of each instance of a complex entity according to the direct storage model and storing the other part according to the normalized storage model [17].

As an example, for the data about the instances of the single complex entity (consisting of the entities "*Composer*", "*Composition*", and "*Post*") described in our example of database, the global relational schema specified in Figure 4 defines their organization in terms of relations that meet the requirements of the normalized storage model.

As an example, the content of Figure 5a represents the transformation of this global relational schema into a global nested relational schema. In this transformation, data about the instances of the complex entity "*Composer*", including instances of its sub-entities "*Post*" and "*Composition*", have been grouped into a single nested relation "*Composers_NR*". This transformation meets the requirements of the direct storage model.

As for the content of Figure 5b and Figure 5c, they illustrate two examples of a combination of these two types of storage models.

In the rest of the paper, we consider that it is the storage model in Figure 5b that corresponds to the designer's choice for this complex entity.

The choice of the nested relational model for this reformulation of the global logical schema of the Data Store—for the clarification of the designer's decisions regarding the storage models that can reduce access times to the data related to instances of the complex entities and detached gray entities—is fundamental in our methodological approach.

This data model has been defined as an extension of the relational model to overcome the limits of its capability to model data describing instances of complex entities [19, 20]. The theoretical foundations of the nested relational model generalize those of the relational model by considering it as a particular case. This makes it possible to base on very well-established common theoretical foundations, the global

relational schema of the Data Store and the global logical schema that results from its reformulation using the nested relational model. In relational model and in nested relational model, the engineering of the schemas of the value collections is based on these theoretical foundations.

(a)

Composers_NR (ComposerId, ComposerName, ComposerBirthDate, ComposerBirthPlace, IsMentoredBy+,
Compositions_R (CompositionId, Title, Duration),
Posts_R (PostId, PostCreationDate, PostSource, PostLenght, PostContent, PostParentIs+))
Concerts_NR (ConcertId, ConcertDate, ConcertPlace)
Programs_NR (ComposerId CompositionId+, ConcertId+)

(b)

Composers_NR (ComposerId, ComposerName, ComposerBirthDate, ComposerBirthPlace, IsMentoredBy+,
Posts_R (PostId, PostCreationDate, PostSource, PostLenght, PostContent, PostParentIs+))
Compositions_NR (ComposerId+, CompositionId, Title, Duration)
Concerts_NR (ConcertId, ConcertDate, ConcertPlace)
Programs_NR (ComposerId CompositionId+, ConcertId+)

(c)

Composers_NR (ComposerId, ComposerName, ComposerBirthDate, ComposerBirthPlace, IsMentoredBy+,
Posts_R (PostId, PostCreationDate, PostSource, PostParentIs+))
Posts_Details_NR (ComposerId, PostId, PostLenght, PostContent)
Compositions_NR (ComposerId+, CompositionId, Title, Duration)
Concerts_NR (ConcertId, ConcertDate, ConcertPlace)
Programs_NR (ComposerId CompositionId+, ConcertId+)

Figure 5. *Examples of Global Nested Relational Schemas Designed for Access Time Optimization.*

In addition, the nested relational model provides normal form conditions for nested relations, which serve as criteria for grouping atomic attributes into relational-valued attributes that nest within each other [20, 21]. These grouping criteria help identify any situation that, based on the business rules, could lead to storage anomalies. Therefore, they provide an additional tool to assist the designer in creating nested relational schemas with good properties.

Let "*GLschema*" denote the global schema of the multi-model database that results from the reformulation of the conceptual schema using the nested relational model and "*TabsSS*", an abstract type such as XML or JSON whose values are flexible-schema or schema-less collections of structured, hierarchical-and-structured or hierarchical-and-semi-structured values.

The previous observations show that the nested relational model allows the designer to ensure that, in "*GLschema*", both structured value collections (adhering to the relational model) and hierarchical-and-structured value collections (adhering to the nested relational model) are described using nested relational schemas designed to: (1) enable the DBMS to ensure database integrity; (2) prevent storage anomalies during creation, modification, and deletion operations; (3) minimize access times for applications to the strict minimum.

The designer can also ensure that, within these nested relational schemas, hierarchical-and-semi-structured value collections (for which the nested relational model does not provide the required flexibility for their description) are treated as atomic abstract values of type "*TabsSS*". This ensures compliance with the constraints of the nested relational model.

As a result, the nested relational model is the most suitable choice to support the modeling of the integration, in a single NewSQL database, of heterogeneous collections of values that adhere to different models. It allows to derive, from the conceptual schema, a global logical schema "*GLschema*" where the nested relational schemas specify value collections that describe the instances of the simple, complex and gray entities of this conceptual schema, on the one hand by taking into account the choices of the designer with respect to the storage models about the complex entities and the detached gray entities and on the other hand, by considering the collections of hierarchical-and-semi-structured values characterized by the irregularity and/or by the unpredictability of their logical structure as being atomic abstract values of type XML or JSON.

As a result, the nested relational model is also the most suitable choice to serve in our methodological approach as a pivot model (c.f. sections VI and VII) for: (1) transforming relational schemas into a NoSQL schemas; (2) unifying design approach of various database types (relational and NoSQL); (3) unifying the mechanisms for processing cross-model queries formulated using a language that integrates into SQL various query languages (object-relational, XML and JSON).

5. Customizing the Physical Database

As highlighted in the introduction, one of the main objectives of NewSQL DBMSs is to: (1) ensure query execution times that meet user requirements; (2) guarantee unlimited horizontal scaling according to workload demands; (3) ensure

uninterrupted data availability.

In existing NewSQL DBMS, achieving this objective depends on the choices made by the designer at the physical schema level of the Data Store, i.e., how data is organized on the physical storage devices. In what follows, the key design choices for a Data Store adhering to the relational model related to sub-objective (1) are addressed in paragraphs 5.1 to 5.6. The choices concerning sub-objectives (2) and (3) are discussed in paragraph 5.7.

5.1. Row Storage Model

The purpose of a query execution engine in a NewSQL DBMS, optimized for online transaction processing (OLTP), is to enable the concurrent execution of a large number of short-duration transactions involving the creation, reading, and updating of a small number of interdependent rows.

To ensure table integrity, each of these I/O operations require access to all columns of each row. This necessitates that the designer make choices that lead the DBMS to physically store each involved table following the row storage model, i.e., the storage of the rows of each table within the physical blocks of the file on the disk dedicated to it [22].

5.2. Columnar Storage Model

The purpose of a query execution engine in a NewSQL DBMS, optimized for online analytical processing (OLAP), is to enable the parallel execution of a large number of interactive analytical processing procedures on massive datasets.

One of the main characteristics of these analytical procedures is that they involve only a small number of columns from the relevant tables and primarily perform read, sort, and aggregation operations on them. The data sources involved may be internal (historical operational data systematically and automatically collected from transactional processes) and/or external.

To reduce the cost of operations performed on these tables, since they typically involve only a small subset of columns, the designer must make choices that lead the DBMS to physically store each involved table according to the columnar storage model, where the data of each column are stored within the physical blocks of a file on the disk dedicated to this column [22].

5.3. Implementing the Direct Storage Model for Complex Entities Using a Table Cluster

Regarding the direct storage model, we previously decided in Section IV to logically store the data about instances of a complex entity into a nested relational table where each row contains the data about a single instance of this complex entity including the data of all the instances of the sub-entities that belong to it.

The physical implementation of this nested relational table can be efficiently carried out in the Data Store within a table cluster [23] that allows us to store in the same physical data block the data about each instance of this complex entity with the data of all the instances of the sub-entities that belong to it.

In this implementation, the tuples containing the data about an instance of this complex entity and about its instances of the sub-entities that belong to it should be grouped and stored in a tree structure, as described in [14], within a physical data block of this cluster of tables, following the hierarchical data organization principles of the nested relational model, i.e., the embedded data approach.

By doing so, at the physical level, the application of algebraic operators on data about the instances of a complex entity stored using the direct storage model will be interpreted as an application of implicit algebraic operators on data stored in main memory.

5.4. Table Indexes

Indexing a table aims for quick access to its rows. It creates an organization of the data in this table on the physical storage medium that allows for each value of a search key to determine the physical addresses of the rows that contain this value, without having to traverse the entire table. Indexing a table thus enables associative searches.

The search key can consist of one or multiple columns corresponding to the primary key, an alternate key, or arising from application needs. The storage of the data related to an index occurs in a physical file distinct from the physical file used for storing the data of this table. However, the index on the primary key can be clustered, meaning it is stored in the same physical file as the table.

Therefore, indexing the tables is a means by which a database designer can enable the DBMS to access all data relevant to each query, using the least amount of time possible, regardless of the complexity of the query and the size of the tables involved [24].

5.5. Join Indexes

The creation of a join index between two tables '*R*' and '*S*' aims to perform repetitive join operations of these two tables based on a join condition defined on their columns, using the least amount of time possible.

If '*r*' denotes the system identifier of the rows in '*R*' and '*s*' denotes the system identifier of the rows in '*S*', the join index would correspond to the highly reduced table obtained by projecting the join operation result of the two tables '*R*' and '*S*' on the pair <*r*, *s*> and sorting the result against these two columns.

The creation of the join indexes is therefore a means by which a database designer can allow the DBMS to execute repetitive join operation as efficiently as possible, such as those resulting from associations identified in the conceptual schema of the database [25].

5.6. Materialized Views

A materialized view consists of: (1) a view based on one or more tables; (2) an internal table whose content corresponds to the result of executing the query that defines this view. This query can be a join query and/or an aggregation query. The content of the internal table can be refreshed by DBMS according to the criteria defined by the designer (e.g., each time a change occurs in the tables used for its computation, at a certain frequency, etc.).

A materialized view can therefore be used for selective replication in a denormalized table of the data from the relational Data Store. As such, the materialized views constitute a means that the designer can use to reduce the execution time of analytical queries by retrieving and grouping together in a single table all the data they need [26].

5.7. Database Sharding

The sharding of a database aims to ensure: (1) unlimited horizontal scaling according to workload demands; (2) uninterrupted data availability [5-12].

A sharded database is manipulated by applications as if it were a centralized database, while it is automatically distributed across a pool of databases called shards. These shards are implemented on separate database servers and may even be geographically distributed to ensure the goals of sharding are met.

The sharding of a database is implemented using a middleware that acts as a proxy. This middleware provides various services, including query filtering and routing, caching of query results, monitoring and rebalancing shard servers, and dynamically adjusting the number of shards based on the workload.

The role of this middleware is twofold: (1) automatically and transparently manage the horizontal partitioning of the tables and their indexes, as well as the distribution of resulting fragments and their replicas across shards; (2) monitor the shards, balance their workload, scale their number up or down depending on workload variations, route queries, and coordinate the distributed and concurrent execution of transactions while ensuring Atomicity, Consistency, Isolation, and Durability (ACID properties.

The query evaluation model for a sharded database involves transferring each elementary operation to the shard containing the required data while maximizing parallelism at both the transaction execution level and the elementary operation execution level.

To achieve optimal performance, the designer must make strategic choices that also minimize network communication overloads. In this query evaluation model, this objective can be met by making the middleware to adopt fragmentation and distribution strategies that ensure, as much as possible, the collocation of joinable rows within the same shard.

An example of a such choice is the implementation of the direct storage model for complex entities by using table clusters. Choosing the direct storage model for detached grey entities is also another example.

6. Transforming Into Global External Logical Schemas

Transforming into external logical schemas takes place only after: (1) the conceptual schema of the database has been specified; (2) the global relational schema of the Data Store and the global nested relational schema, which clarifies the designer’s choices regarding storage models for complex entities and detached gray entities have been designed consistently; (3) the physical schema of the Data Store has been designed to ensure performance, scaling, and fault tolerance.

In other words, this transformation occurs only after creating all the tables of the Data Store incorporating all decisions made at the physical schema level.

In the ANSI/SPARC architecture of the schemas, shown in Figure 1, global external logical schemas redefine the Data Store as a multi-model database. This means that the Data Store can be viewed and manipulated by developers, application integrators, and data analysts as if it were, virtually, a relational, object-relational, XML, JSON, or graph-oriented database. This approach provides them with the agility and flexibility needed to work with their preferred data models [14] and unifies the design methods of various NoSQL database types.

From the designer's point of view, redefining the Data Store as a virtual object-relational, XML, or JSON database follows a different approach than that about virtual graph-oriented databases. The following presents these two approaches.

6.1. Redefining as a Virtual Database (Object-relational, XML, or JSON)

Let "*GLPschema*" be the global logical schema derived from "*GLschema*" (as defined in Section IV) by integrating access paths that can meet user needs.

Figure 6 provides, as an example, the global logical schema derived from the global logical schema in Figure 5b by integrating access paths that can be inferred from the conceptual schema in Figure 3.

The design objectives of "*GLPschema*" are to: (1) enable the DBMS to ensure database integrity; (2) prevent storage anomalies; (3) enable fast access to data about complex real-world entities; (4) provide users with the access paths that can meet their needs.

In our methodological approach, the role of "*GLPschema*" is to serve as a global nested relational pivot schema.

To allow users to work with their preferred data models (object-relational, XML, and/or JSON), the designer must derive, for each of these models, the global external schema of a virtual database from the same underlying global nested relational pivot schema, namely "*GLPschema*".

This involves: (1) creating the user-defined object-relational type, XML schema, or JSON schema that specifies the structural integrity constraints for each typed view; (2) providing the DBMS with the necessary specifications to dynamically handle the "*relational model ↔ non-relational model*" data mapping, enabling the manipulation of typed views in virtual databases whenever needed.

The creation of a user-defined object-relational type, an XML schema, or a JSON schema, which defines the structure of instances of a typed view in a virtual database, as well as the generation of the logical execution plan for the dynamic derivation of the instances of this typed view from the relevant data in the Data Store, can result from an automatic transformation process of the required underlying nested relational pivot schema defined in "*GLPschema*".

To achieve this, the designer must define, if necessary, for atomic-valued attributes in "*GLPschema*", specific rules for a customized automatic transformation of the nested relational schemas.

These specific rules can be used, for example, to indicate that: (1) the atomic-valued attributes "*ZipCode*", "*Town*", and "*Country*" must be grouped into a composite-valued attribute "*Address*"; (2) the atomic-valued attribute "*PartNumber*" must be treated in an XML document as an XML element rather than as an XML attribute.

```
Composers_NRP (ComposerId, ComposerName, ComposerBirthDate, ComposerBirthPlace, IsMentoredBy+,
               Posts_R (PostId, PostCreationDate, PostSource, PostLenght, PostContent, PostParentIs+),
               IsMentorOf (ComposerId+),
               HasProduced (CompositionId+))
Compositions_NRP (ComposerId+, CompositionId, Title, Duration,
                  IsProgrammedIn (ConcertId+))
Concerts_NRP (ConcertId, ConcertDate, ConcertPlace,
              HasProgrammed (ComposerId CompositionId+))
Programs_NRP (ComposerId CompositionId+, ConcertId+)
```

Figure 6. *Example of a Global Nested Relational Pivot Schema, Derived From the Schema in Figure 5b By Integrating Access Paths, Underlying the NoSQL Virtual Databases.*

Alternatively, the designer also has the option to manually define the schema of each typed view and to create the SELECT statement required to generate the logical execution plan for the dynamic derivation of instances of this typed view from relevant data in the Data Store.

The automatic generation of the logical execution plan for the derivation of the instances of a non-relational typed view (object-relational, XML, or JSON) based on the transformation rules can be seen as a two-step data mapping process.

6.1.1. The First Step of the Data Mapping Process

The first step is identical for all three models (object-relational, XML, and JSON). This step involves the application of the transformation rules on the required underlying nested relational pivot schema to derive a sequence of nested relational algebra operations. The purpose of this sequence is to define a logical execution plan for the generation of the tuples of a table that adheres to this required underlying nested relational pivot schema. The tables used for generating these tuples must be the tables of the Data Store.

Figure 7 contains an example that concerns a typed view (object-relational, XML or JSON) containing the data about the instances of the complex entity "*Composer*" including the data about the instances of its sub-entity "*Post*" and the data about access paths. The content of this figure is the logical execution plan that could be generated from the underlying nested relational schema of Figure 6, namely "*Composers_NRP*", based on the transformation rules.

The purpose of this logical execution plan is to generate the tuples of a table that adheres to "*Composers_NRP*", using the tables "*Composers_R*", "*Posts_R*", and "*Compositions_R*" of the Data Store, as described in Figure 4.

Each tuple generated by this logical execution plan is about a composer. It consists of the atomic attributes "ComposerId", "*ComposerName*", "*ComposerBirthDate*", "*ComposerBirthPlace*", "*IsMentoredBy*" and the relational-valued attributes "*Posts_R*" (containing all the posts that concern this composer), "*Is_MentorOf*" (containing as access paths all the identifiers of the composers he mentors), and "*Has_Produced*" (containing as access paths all the identifiers of the productions he authored).

As for Figure 8, it contains the logical execution plan that could be generated from the underlying nested relational schema of Figure 6, namely "*Compositions_NRP*", for a typed view (object-relational, XML or JSON) containing the data about the instances of the entity "*Composition*" including data about the access paths, using the tables "*Compositions_R*" and "*Programs_R*" of the Data Store (see Figure 4).

6.1.2. The Second Step of the Data Mapping Process

As for the second step of the data mapping process, it involves transforming each logical execution plan generated during the first step into a logical execution plan that produces all the instances of the relevant non-relational typed view.

This can be done by relying on the transformation rules to apply the appropriate transformation functions (defined for this purpose in the ISO standards) to the results of operations or groups of operations in each logical execution plan produced in the first step.

The structure of each instance produced in this second step must adhere to the schema of the concerned typed view.

```
π [c1.ComposerId, c1.ComposerName, c1.ComposerBirthDate, c1.ComposerBirthPlace, c1.IsMentoredBy,
   π [p.PostId, p.PostCreationDate, p.PostSource, p.PostLenght, p.PostContent, p.PostParentIs]
     σ [p.ComposerId = c1.ComposerId] (Posts_R p) : Posts_R,
   π [c2.ComposerId] σ [c2.IsMentoredBy = c1.ComposerId] (Composers_R c2) : IsMentorOf,
   π [c3.CompositionId] σ [c3.ComposerId = c1.ComposerId] (Compositions_R c3) : HasProduced
  ] (Composers_R c1)
```

Figure 7. *Logical execution plan for generating the rows adhering to the nested relational pivot schema "Composers_NRP" underlying a typed view regardless of its model.*

```
π [c.ComposerId, c.CompositionId, c.Title, c.Duration,
   π [p.ConcertId] σ [p.ComposerId = c.ComposerId ∧ .p.CompositionId = c.CompositionId] (Programs_R p) :
     IsProgrammedIn
  ] (Compositions_R c)
```

Figure 8. *Logical execution plan for generating the rows adhering to the nested relational schema "Compositions_NRP" underlying a typed view regardless of its model.*

For example, Figure 9 contains three logical execution plans. The transformation functions shown in plans (2) and (3) are functions of the ORACLE'S DBMS, used for illustration purposes.

Plan (1) is extracted from the logical execution plan in Figure 7, derived during the first step of the data mapping process from an underlying nested relational pivot schema, regardless of the data model of the concerned view. This plan computes the value of the relational-valued attribute "*HasProduced*" for each tuple generated by the logical execution plan in Figure 7. This value is a relation consisting of a single attribute, "*CompositionId*", meaning it is a set of values for this attribute corresponding to access paths. Each value must correspond to a foreign key referencing a tuple in the "*Compositions_R*" relation.

Plan (2) is the result of transforming Plan (1) during the second step of the data mapping process for an object-relational virtual database. The "*MAKE_REF(T, a)*" function converts the foreign key value "*a*" into a logical pointer to the referenced row in table "*T*". The composite function "*CAST(MULTISET(SubQuery) AS NewType)*" converts the collection type produced by the subquery "*SubQuery*" into a user-defined type ("*ProducedProductionList*"), which defines nested tables.

As for plan (3), it is the result of the transformation of plan (1) carried out during the second step of the data mapping process when dealing with an XML virtual database. The function "*XMLELEMENT (ElmtName, ElmtValueExpr)*" replaces the result of evaluating the expression "*ElmtValueExpr*" with an XML element named "*ElmtName*" with a body equal to the result of evaluating the expression "*ElmtValueExpr*". The composite function "*XMLAGG(SubQuery XMLELEMENT(ElmtName, ElmtValueExpr))*" replaces the collection resulting from the execution of the subquery "*SubQuery*" passed as a parameter with a collection of XML elements, each named "*ElmtName*" with a body equal to the value of the projection of the attribute "*CompositionId*" on a row of the result of this subquery "*SubQuery*" passed as a parameter.

6.2. Redefining as a Virtual Graph-Oriented Database

As a reminder, in the global logical schema of a relational database, foreign keys result from the materialization of associations defined in its conceptual schema. In the database, each foreign key value establishes a semantic link (i.e., a relationship) between two instances of one or two real-world entities.

Thus, it is possible to infer from the data stored in an operational relational database, the set of semantic links that exist between the instances of the real-world entities described in it.

```
(1) π [c3.CompositionId] σ [c3.ComposerId = c1.ComposerId] (Compositions_R c3) : HasProduced

(2) CAST(MULTISET(π [MAKE_REF(c3, c3.CompositionId)]
                    σ [c3.ComposerId = c1.ComposerId] (Compositions_R c3)) : HasProduced
    AS ProducedProcductionList

(3) XMLELEMENT (
          "HasProduced",
          (XMLAGG (σ [c3.ComposerId = c1.ComposerId] (Compositions_R c3)
                XMLELEMENT ("CompositionId", π [c3.CompositionId])))
```

Figure 9. *Examples of Applying Transformation Functions to Logical Execution Plans.*

The graph-oriented data model enables the representation of these semantic links using a graph. In this graph, each vertex corresponds to an instance of a real-world entity described in the database. Each edge represents a semantic link between two instances of one or two real-world entities, as established in the database through the value of a foreign key. Each vertex and edge is defined by a label that determines its type and by a set of properties.

The ISO SQL/PGQ (Property Graph Query) standard [27] defines a set of features that enable the materialization of a graph using tables of vertices (or nodes) and tables of edges (or semantic links) where each element corresponds to a row in an operational relational database table. Just like views, these tables of vertices and these tables of edges do not have a real existence; they exist only through their definitions. Each of their elements is dynamically determined whenever needed, based on the corresponding row in the operational relational database. This "*relational model → graph model*" data mapping is handled by the DBMS, relying on the specifications provided by the designer in the graph metadata.

Figure 10 provides an example of an SQL statement for creating a graph titled "*composers_graph*", derived from the "*Composers_R*" table defined in the relational logical schema of our example database, as described in Figure 4. This statement specifies that this graph will be materialized in the vertex table "*Composers_R*" and the edge table "*Composers_R*".

Furthermore, the features specified in the ISO SQL/PGQ standard [27] allow developers to formulate an SQL pattern-matching query on a graph created in this way, enabling them to discover all subgraphs that match the pattern defined by this query.

The purpose of the query language defined by this standard is to integrate graph manipulation into SQL, while making the specification of computing operations more compact and less complex than in approaches based on the explicit use of the relational join operator.

The integration of graph processing within the "*SELECT*" statement is primarily achieved through the use of the "*GRAPH_TABLE*" operator in the "*FROM*" clause, along with the "*MATCH*" and "*COLUMNS*" clauses. This operator and these clauses serve two purposes: (1) defining a pattern for graph pattern matching; (2) specifying a virtual table to store the matching results.

A "*FROM*" clause containing such specifications can also include non-graph-oriented tables required for evaluating the "*SELECT*" statement. This makes it possible to formulate cross-model queries that combine tables from different data models, including the graph-oriented model.

Figure 11 contains an example of a pattern-matching SQL query on the graph whose creation statement is shown in Figure 10. This query returns the list of composers who have a mentor, indicating for each composer his name and the name of his mentor.

```
CREATE PROPERTY GRAPH composers_graph
   VERTEX TABLES (
      Composers_R KEY (ComposerName)
        LABEL composer
        PROPERTIES (ComposerName, ComposerBirthPlace, IsMentoredBy))
   EDGE TABLES (
      Composers_R AS IsMentoredBy
        SOURCE KEY (ComposerName) REFERENCES Composers_R (ComposerName)
        DESTINATION KEY (IsMentoredBy) REFERENCES Composers_R (ComposerName
        LABEL MentoredBy
        NO PROPERTIES);
```

Figure 10. *Example of SQL statement for creating a graph.*

```
SELECT ComposerName, MentorName
   FROM GRAPH_TABLE (composers_graph
      MATCH
         (c1 IS composer) -[e IS MentoredBy]-> (c2 IS composer)
      COLUMNS (c1.ComposerName AS ComposerName, c2.ComposerName AS MentorName));
```

Figure 11. *Example of SQL pattern-matching query on a graph.*

The redefinition of the Data Store as a virtual graph-oriented database therefore consists for the designer in creating a set of graphs using vertex tables and edge tables that do not have real existence, where each element corresponds to a row of a table of the Data Store. The identification of these graphs must result from an analysis of the enterprise's needs in terms of analysis of the semantic links existing between the instances of the real-world entities described in the Data Store.

7. Query Processing in the Multi-Model Database

A multi-model database, stored within a Data Store adhering to relational model, optimized for hybrid processing (both transactional and analytical), and accessed by developers and data analysts through relational and non-relational virtual databases (object-relational, XML, JSON) defined via external schemas, provides exceptional flexibility to its users.

They can manipulate it freely and with no overload as: (1) a relational database; (2) a non-relational database (object-relational, XML, JSON); (3) a hybrid database consisting of typed views that conform to different data models (relational, object-relational, XML, JSON).

In other words, the user can formulate a cross-model SQL query that manipulates typed views of different types (relational, object-relational, XML, JSON), potentially containing textual values that correspond to flexible-schema or schema-less XML or JSON data, which are natively processed using the query language of their respective model.

In NewSQL DBMS, the manipulation of NoSQL values within SQL is largely achieved using query functions defined by the ISO standards such as "*SQL3*", "*SQL/XML*" and "*SQL/JSON*". These standards aim to ensure interoperability between SQL and NoSQL query languages.

These query functions typically take two input parameters: (1) in on hand, a list of values, where each value corresponds to either a table row or an instance of a NoSQL typed view being processed, or a NoSQL value contained within a table row or within an instance of a NoSQL typed view being processed; (2) in the other hand, the NoSQL query to be applied to the first parameter.

The role of the NoSQL query is to extract values from the NoSQL values within the first parameter.

When used in a "*SELECT*" clause, the query functions enable results that include columns of different types (scalar types, user-defined object-relational types, XML types, JSON types). These columns may come from virtual tables of different types (relational, object-relational, XML, JSON) cited in the "*FROM*" clause.

These query functions can also serve as selection test functions in the "*WHERE*" clause.

As mentioned in Section VI, in our approach, the NoSQL virtual databases that allow users to manipulate the Data Store as both an object-relational, XML, and JSON database share the same underlying global nested relational pivot schema, namely "*GLPschema*", from which they were derived. Let "*Si*" be a schema within "*GLPschema*". The logical execution plan derived from "*Si*" (as indicated in Section VI) for generating instances of a typed view is identical for all three models (object-relational, XML, and JSON) when model-specific transformation functions are ignored (see as example Figure 9).

The processing of cross-model SQL queries can therefore involve the DBMS treating in a first step these queries as if they are formulated on tables adhering to the underlying nested relational pivot schemas, regardless of the model of the typed views involved. These underlying pivot schemas may contain string-type attributes that hold XML or JSON documents, which can be manipulated using the query language specific to their model.

This section illustrates, using an example, this approach that enables the unification of cross-model SQL query processing, specifically by showing how to transform these queries into optimized logical execution plans.

Figure 12 presents an example of a cross-model SQL query that takes as input two object-relational typed views, "*Composers_OR*" and "*Compositions_OR*". As a reminder, the logical execution plans presented in Figures 7 and 8 correspond respectively to these two typed views, regardless of the model type to which they adhere, which could also have been XML or JSON. These logical execution plans were generated to derive instances of the tables that adhere to their underlying nested relational schemas in Figure 6, namely, "*Composers_NRP*" and "*Compositions_NRP*".

The result of this cross-model query is a tuple consisting of two atomic-valued attributes ("*Title*": the title of a composition with the identifier "*200*", "*ComposerName*": the name of the composer) and one relational-valued attribute ("*PostedToComposer*": a list of posts related to this composer, restricted to the columns indicating the creation date and source).

In this cross-model SQL query, the transformation function "*TABLE*" is used to convert the content of the relational-valued attribute "*Postes_R*" (from the instance of the typed view "*Composers_OR*" being processed) into a relational table.

```
SELECT c4.Title, c1.ComposerName,
        (SELECT PostCreationDate, PostSource FROM TABLE(c1.Posts_R)) AS PostedToComposer
FROM Composers_OR c1, Compositions_OR c4
WHERE c1.ComposerId = c4.ComposerId AND c4.CompositionId = 200;
```

Figure 12. *Example of a Cross-Model SQL Query.*

Figure 13 presents the gross logical execution plan that the DBMS could derive from this cross-model SQL query.

```
π [c4.Title, c1.ComposerName,
    π [p2.PostCreationDate, p2.PostSource] (c1.Posts_R p2) : PostedToComposer]
(Composers_OR c1 ∞ (c1.ComposerId = c4.ComposerId AND c4.CompositionId = 200) Compositions_OR c4)
```

Figure 13. *Gross Logical Execution Plan for Query in Figure 12.*

To enable the evaluation of this execution plan on the Data Store, it is sufficient to replace the two object-relational typed view with the logical execution plans shown in Figures 7 and 8 as if this query was formulated on the tables adhering to the underlying nested relational pivot schemas of these two typed views, namely, "*Composers_NRP*" and "*Compositions_NRP*".

Figure 14 shows the resulting execution plan. In this execution plan, the object-relational typed views "*Composers_OR*" and "*Compositions_OR*" have been replaced by the three relational tables ("*Composers_R*", "*Posts_R*", and "*Compositions_R*") used to compute their instances.

Simplifying the logical execution plan of Figure 14 leads to the execution plan of Figure 15.

As discussed in Section II, "*Composer*" is a complex entity with "*Post*" as its sub-entity. The global nested relational schema in Figure 5b indicates that the designer chose a hybrid storage model for this complex entity. As a result, the relational tables "*Composers_R*" and "*Posts_R*", used in this logical execution plan, must be considered part of a table cluster. This means that: (1) the data about each composer and the posts concerning him are stored in the same physical block of the disk; (2) the relational algebraic operators involved in processing these data must consequently be interpreted as implicit relational algebraic operators, which the DBMS evaluates in main memory.

```
π [c4.Title, c1.ComposerName,
    π [p2.PostCreationDate, p2.PostSource] (c1.Posts_R p2) : PostedToComposer]
((π [c1.ComposerId, c1.ComposerName, c1.ComposerBirthDate, c1.ComposerBirthPlace, c1.IsMentoredBy,
     π [p2.PostId, p2.PostCreationDate, p2.PostSource, p2.PostLenght, p2.PostContent, p2.PostParentIs]
       σ [p2.ComposerId = c1.ComposerId] (Posts_R p2) : Posts_R,
     π [c2.ComposerId] σ [c2.ComposerId = c1.ComposerId] (Composers_R c2) : IsMentorOf,
     π [c3.CompositionId] σ [c3.ComposerId = c1.ComposerId] (Compositions_R c3) : HasProduced
    ] (Composers_R c1))
∞ (c1.ComposerId = c4.ComposerId AND c4.CompositionId = 200)
(π [c4.ComposerId, c4.CompositionId, c4.Title, c4.Duration, ,
    π [p3.ConcertId] σ [p3.ComposerId = c4.ComposerId ∧ .p3.CompositionId = c4.CompositionId]
      (Programs_R p3) : IsProgrammedIn
   ] (Compositions_R c4)))
```

Figure 14. *Gross Logical Execution Plan for Evaluating Query in Figure12 on the Data Store.*

```
π [c4.Title, c1.ComposerName, PostedToComposer]
((π [c1.ComposerId, c1.ComposerName,
     π [p2.PostCreationDate, p2.PostSource]
       σ [p2.ComposerId = c1.ComposerId] (Posts_R p2) : PostedToComposer
    ] (Composers_R c1))
∞ (c1.ComposerId = c4.ComposerId)
(π [c4.ComposerId, c4.Title]
  σ [c4.CompositionId = 200] (Compositions_R c4)))
```

Figure 15. *Optimized Logical Execution Plan for Evaluating the Query in Figure12 on the Data Store.*

Therefore, having in addition a join index between "*Composers_R*" and "*Compositions_R*" would allow this cross-model query to be executed in the shortest possible time.

Figure 16 shows a transformation, in a second step, of the logical execution plan from Figure 15, aiming to generate a result where each column's type matches the type inferred from the initial cross-model query. In this transformation, the type of the relational-valued attribute "*PostedToComposer*" has been converted into an Oracle user-defined nested table type, namely "*PostedToComposerList*", using the "CAST-MULTISET" transformation functions.

```
π [c4.Title, c1.ComposerName,
   CAST(MULTISET(PostedToComposer]) AS PostedToComposerList) : PostedToComposer
((π [c1.ComposerId, c1.ComposerName,
     π [p2.PostCreationDate, p2.PostSource]
       σ [p2.ComposerId = c1.ComposerId] (Posts_R p2) : PostedToComposer
    ] (Composers_R c1))
∞ (c1.ComposerId = c4.ComposerId)
(π [c4.ComposerId, c4.Title]
   σ [c4.CompositionId = 200] (Compositions_R c4)))
```

Figure 16. *Application of Transformation Functions on the Optimized Logical Execution Plan for the Final Rendering of the Evaluation of the Query in Figure12 on the Data Store.*

Figures 17 and 18 highlight how the processing of a path expression should be handled in our approach, using the object-relational model as an example.

In Figure 17, "*IsProducedBy*" results from the transformation of the foreign key "*ComposerId*" (defined in the underlying nested relational schema "*Compositions_NRP*" from Figure 6) into a logical pointer in the object-relational model.

Finally, as discussed in Section VI, the "FROM" clause in a cross-model SQL query that manipulates views of different types (relational, object-relational, XML, JSON) can also include the specification of a table that contains the result of a graph pattern matching operation. This possibility allows SQL cross-model queries supporting graph manipulation. To ensure that the evaluation of such queries follows the same approach as the one illustrated, it requires the use of a new relational algebraic operator, namely the MAP operator [28].

```
SELECT c1.Title, c1.IsProducedBy.ComposerName AS ComposerName FROM Compositions_OR c1;
```

Figure 17. *Query using a path expression.*

```
π [c1.Title,
   π [c2.ComposerName] σ [c2.ComposerId = c1.ComposerId] (Composers_R c2) : ComposerName
] (Compositions_OR c1)
```

Figure 18. *Gross Logical Execution Plan for the Query in Figure 17.*

8. A Framework for Supporting the Design Process

A data model can be viewed as a virtual machine characterized by four sets: (1) a set of value types; (2) a set of structure types to which complex values can adhere; (3) a set of operation types enabling the manipulation of values; (4) a set of integrity constraint types that can be enforced by the DBMS.

If M1 and M2 represent two different data models, the mapping of a schema S1 from M1 to M2 consists of redefining (i.e., transforming) S1 into a schema S2 in M2 that is equivalent to S1.

The transformation mechanisms for mapping a schema from one model to another have been extensively studied, both theoretically [29] and practically [30], for various applications.

These mechanisms can be integrated into a graphical tool within a database-oriented application design and development platform to automate a significant part of the design methodology presented in this paper, which is primarily based on a schema mapping process.

The role of such a tool can be as follows:

Assist the designer in drawing a conceptual schema free of semantic heterogeneity, where simple entities, complex entities, and gray entities are properly modeled.

Assist the designer to transform the conceptual schema into a global relational logical schema of the Data Store and a global nested relational logical schema, ensuring compliance with the rules defined in Sections III and IV.

Guide the designer in making choices regarding the physical layout of the Data Store.

Create in the database dictionary, under the designer's supervision, the tables of the Data Store and their indexes, incorporating all design choices.

Generate in the dictionary of the database, with the designer's assistance, all the artifacts required for the creation of the virtual databases: underlying nested relational pivot schemas including access paths, annotated user-defined types, annotated XML schemas, annotated JSON schemas, typed views and their execution plans for deriving their instances.

9. Related Work

The approach presented in this paper allows an assessment

of the capability of NewSQL DBMSs to meet the objectives for which they were introduced, namely, to offer the advantages of NoSQL DBMSs while avoiding their drawbacks and preserving the benefits of SQL DBMSs. This presentation also allows for a comparison of the advantages of NewSQL DBMSs with those provided by other types of approaches.

This section successively broach: (1) the strengths and the weaknesses of NewSQL DBMSs in achieving their objectives, regarding their capabilities highlighted in the approach we described; (2) a brief comparison with the approach that involves evolving existing NoSQL DBMSs into their multi-model versions; (3) a brief comparison with the approach that aims the creation of a new model that natively integrates the capabilities of several models; (4) a brief comparison with multistore system approaches; (5) a brief comparison with NoSQL Database Design Methods.

A comprehensive overview of the various existing approaches can be found in [1, 2, 31].

9.1. Strengths of NewSQL DBMSs

Digital transformation has made databases one of the key foundations upon which modern societies are built. They serve as the memory that enables societies to establish good governance, simplify the lives of all stakeholders, drive innovation across industries, increase productivity, and reduce production and service costs.

To achieve this, a database must be versatile and application-independent, a guarantor of reliability and accessibility, ensuring the accuracy of the information it stores and the insights and predictions it enables, robust while maintaining a straightforward implementation.

NewSQL DBMSs allow the designer to choose the relational model as the primary model for the Data Store.

In the current state of the art, this technological offering represents the best choice for enabling the DBMS to serve as the guarantor of integrity for structured or hierarchical-and-structured value collections (adhering to the relational or nested relational models), where this aspect is considered mission-critical by users.

This technological offering also enables, within a column of a relational table, the native storage of value collections that adhere to XML or JSON standards. For these types of data, the primary requirement is not integrity but rather their highly flexible format, which allows them to be self-documenting, complex, hierarchical, and semi-structured.

It also enables the designer to provide developers, application integrators, and data analysts with the agility and flexibility needed to seamlessly interact with the relational Data Store, without overload, as if it were, virtually a relational, object-relational, XML, JSON, and graph-oriented database simultaneously.

The integration of object-relational, XML, JSON, and graph-oriented query languages into SQL ensures that, within each cross-model query, each value can be processed using the query language of its respective model.

In other words, this technological offering provides the possibility of ensuring, using a single DBMS, the storage and manipulation of hybrid collections of values adhering to different models as needed.

Furthermore, this technological offering provides the designer with multiple options for organizing data on the physical storage devices used for data persistence. These options can help: (1) minimize query execution time; (2) ensure unlimited horizontal scaling and uninterrupted data availability; (3) coordinate concurrent and distributed transaction execution while guaranteeing the ACID properties.

As a final advantage, it is worth noting that NewSQL DBMSs are extensible to new data models, following the same approach used for the integration of XML, JSON, and the graph-oriented model.

9.2. Weaknesses of NewSQL DBMSs

NewSQL DBMSs also allow the designer to store the data in tables of different types (relational, object-relational, XML, and JSON) that have a real existence. For the NoSQL tables (object-relational, XML, and JSON), this capability is generally based on the integration of their storage engines with the SQL storage engine, enabling either storing NoSQL data in its native format, or automatically shredding NoSQL data into internal tables (relational or object-relational, for example). The goal of this feature is to ensure that the primary model of the Data Store can be any model, not just the relational model.

Leveraging this capability can lead to poorer practices for three main reasons: (1) NoSQL models encourage the designer to define the logical structure of values based on both their semantics and access paths required by developers. In contrast, the relational model requires defining the logical structure of values solely based on their semantics; (2) NoSQL models promote materializing associations using pairs of semantic links, where each link in each pair must be the inverse of the other, while the DBMS has no built-in mechanism to guarantee the integrity of this materialization [14]; (3) the integrity constraints that NoSQL models allow the DBMS to enforce are only a subset of those provided by the relational model.

This second option, related to choosing the primary model for the Data Store, can therefore make the database less versatile and less independent from applications. Furthermore, it may also reduce DBMS's ability to ensure data reliability, potentially forcing developers to handle integrity controls themselves, which is an error's prone task. This can make the database significantly less robust.

Finally, it is important to note that in NewSQL DBMSs, most of the physical database organization options described in Section V apply only to relational tables. This is especially true for database sharding.

9.3. The Evolution of Existing NoSQL DBMSs Towards Their Multi-model Versions

In this approach, existing offerings [1, 2] can only support a limited number of different models.

Except for the possibilities it offers for storing data that adhere to multiple models, the advantages of this approach are the same as those of the original model.

9.4. The Creation of a New Model That Natively Integrates the Capabilities of Several Models

Just like the NewSQL approach this category aims at unifying data models. For example, SQL++ [4] is a semi-structured model that extends the capabilities of the relational model and JSON in terms of modeling possibilities. SQL++ also provides a query language that is fully backward compatible with SQL while integrating the capabilities of semi-structured models.

The NewSQL approach and the SQL++ approach are fundamentally different. The NewSQL approach relies on standards to integrate other models under the relational model. The SQL++ approach starts from scratch to define a new model that incorporates the features of multiple models (i.e. a model composed of several models).

Each approach has its own benefits. The NewSQL approach follows an evolutionary path, ensuring backward compatibility and allowing businesses to gradually adopt its new capabilities. The SQL++ approach takes a revolutionary path, enabling the creation of an entirely new technological offering, leveraging lessons learned and the latest advancements in database technology. Thus, one could say that SQL++ paves the way for new types of lighter and more efficient DBMSs.

However, in terms of modeling capability, NewSQL and SQL++ are potentially equivalent, as both could theoretically allow the integration within a single database of the same types of value collections adhering to same types of models.

9.5. Multistore Systems

The purpose of multistore systems is to provide users (developers, application integrators, and data analysts) with the necessary tools to manipulate heterogeneous collections of values stored in autonomous databases that adhere to different data models (relational and NoSQL). These databases may have been designed independently or simultaneously for predefined needs. Heterogeneity can affect multiple aspects, including: (1) structural differences, such as variations in how the logical structure of data describing the instances of the entities is defined; (2) impedance mismatches between the data types supported by the different types of DBMSs; (2) syntactic differences in query languages; (4) semantic issues, such as the presence of synonyms and homonyms.

Providing the necessary tools for manipulating these data collections requires providing users with a unified global view and a cross-model query language based on this view. To achieve this, various approaches have been proposed, relying on a middleware composed of one mediator and several wrappers, each dedicated to a specific type of DBMS [32-34].

NewSQL DBMS and multistore systems have been designed to meet different objectives. They offer different advantages that are complementary. NewSQL DBMS provide the ability to integrate heterogeneous collections of values that adhere to different types of data models within a single database while ensuring scaling, fault tolerance, availability, and ACID properties (Atomicity, Consistency, Isolation, Durability). Multistore systems, on the other hand, enable the federation of relational, NoSQL, and NewSQL databases implemented on autonomous servers without compromising their autonomy and without impacting their properties. This is a fundamental aspect for enterprises that want to access Big DATA

9.6. NoSQL Database Design Methods

Initially, the implementation of a NoSQL database only focused on the physical level.

The main goal of the design methods of NoSQL databases is to ensure that the design process starts with the functional requirements of users, rather than non-functional requirements (such as performance, scalability, high availability), similar to the methods developed for relational databases.

The main research efforts on this topic have highlighted the possibility of defining unified design methods, independent of the type of NoSQL DBMS, by distinguishing three levels of abstraction: the conceptual level, the logical level, and the physical level [35].

Among the methods based on this approach, "*UMLtoNoSQL*" [36] is one of the most comprehensive.

This approach leverages the MDA (Model-Driven Architecture) paradigm, defined by the OMG (Object Management Group), for the automatic transformation of schemas [30]. It uses: (1) the UML model to define at the conceptual level a UML diagram that adheres to a metamodel; (2) automatic transformation rules at the logical level to convert the UML diagram into a generic logical schema, adhering to a metamodel compatible with three NoSQL models (column-family, document-oriented, or graph-oriented); (3) automatic transformation rules at the physical level to convert the generic logical schema into a NoSQL database (column-family, document-oriented, or graph-oriented).

As for the NewSQL approach presented in this paper, it implements two design processes where each process consider four levels of abstraction. In these two design processes, the global nested relational pivot schema plays a unifying role.

In the "*UMLtoNoSQL*" approach, schema transformations

are performed fully automatically, whereas in the NewSQL approach, related to these two design processes, transformations are mostly semi-automatic to allow the designer to customize the resulting schemas. This customization enables the consideration of user specific requirements.

In addition to schema mapping, the NewSQL approach described in this paper also requires dynamic data mapping at runtime. This means that data conversions occur dynamically, transforming data adhering to one model into data adhering to another model. These conversions take place between the Data Store and the virtual NoSQL databases.

10. Conclusion

The evolution of SQL DBMSs toward NewSQL DBMSs has led to the incorporation of functionalities that provide database designers with a wide range of possibilities, thereby increasing the complexity of database design.

In this paper, we have shown through an example how to design and implement, on a NewSQL DBMS, a database that combines the advantages of both SQL and NoSQL databases by leveraging on: (1) the comprehensive methodological approach that we propose for integrating consistently, by relying on five levels of abstraction, the design process of a relational Data Store and the design process of virtual databases that allow the Data Store to be viewed and manipulated as a NoSQL database (object-relational, XML, JSON or graph-oriented); (2) the possibilities highlighted for data organization on storage devices; (3) the identified possibilities for handling operations on the NoSQL virtual databases using a uniform, innovative, and efficient approach.

The ISO standards for extending the relational model, as well as the capabilities highlighted in our methodological approach as fundamental for data organization on storage devices, have not yet been fully implemented by all SQL DBMSs, particularly with regard to the most recent features such as SQL/JSON and SQL/PGQ standards, table clusters [14], and database sharding.

Currently, database technology is still largely dominated by the relational model [37]. As a result, many businesses face challenges in adapting their mission-critical applications to the web's demands for scalability and data availability. For these companies, NewSQL DBMSs offer significant advantages: (1) the ability to help them overcome their current challenges; (2) backward compatibility, ensuring the continued smooth operation of existing applications; (3) the ability to run on low-cost standard servers, leading to reduced investment and recurring costs; (4) a vast community of designers and web application developers already familiar with most of the core concepts on which NewSQL is based.

The world of SQL DBMSs is currently undergoing active exploration for a transformation that can best meet this new demand from businesses [6, 7].

This presentation enables researchers and technology providers to identify key areas that require special attention to bring significant improvements to NewSQL DBMSs.

Abbreviations

DBMS	Database Management System
SQL	Structured Query Language
NoSQL	Not Only SQL
ISO	International Organization for Standardization
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
PGQ	Property Graph Query
1NF	First Normal Form
ACID	Atomicity, Consistency, Isolation, and Durability
ANSI	American National Standards Institute
SPARC	Standards Planning & Requirements Committee
LOB	Large Object
E/A	Entity/Association
UML	Unified Modeling Language
3NF	Third Normal Form
4NF	Fourth Normal Form
OLTP	Online Transaction Processing
I/O	Input/output
OLAP	Online Analytical Processing
MDA	Model-Driven Architecture
OMG	Object Management Group

Author Contributions

Joachim Tankoano is the sole author. The author read and approved the final manuscript.

Conflicts of Interest

The author declares no conflicts of interest.

References

[1] Qingsong Guo1, Chao Zhang, Shuxun Zhang, Jiaheng Lu. Multi‐model query languages: taming the variety of big data. Distributed and Parallel Databases (2024) 42: 31–71.

[2] Jiaheng Lu and Irena Holubova. Multi-model Databases: A New Journey to Handle the Variety of Data. ACM Computing Surveys. 2019, Vol. 0, No. 0.

[3] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, Fred Zemke. The New and Improved SQL: 2016 Standard. SIGMOD Record. June 2018 (Vol. 47, No. 2).

[4] Kian Win Ong, Yannis Papakonstantinou, Romain Vernoux. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases. arXiv: 1405.3631 [cs.DB], Dec. 2015.

[5] Manjunatha Sughaturu Krishnappa, Bindu Mohan Harve, Vivekananda Jayaram, Akshay Nagpal, Koushik Kumar Ganeeb, Balaji Shesharao Ingole. ORACLE 19C Sharding: A Comprehensive Guide to Modern Data Distribution. IJCET. Sep-Oct 2024, Volume 15, Issue 5.

[6] Samuel Akinola. Trends in Open Source RDBMS: Performance, Scalability and Security Insights. Journal of Research in Science and Engineering (JRSE). July 2024, Volume-6, Issue-7.

[7] Naresh Kumar Miryala. Emerging Trends and Challenges in Modern Database Technologies: A Comprehensive Analysis. International Journal of Science and Research (IJSR). November 2024, Volume 13 Issue 11.

[8] Abdullah Muhammed, Zul Hilmi Abdullah, Waidah Ismail1, Ali Y. Aldailamy, Abduljalil Radman, Rimuljo Hendradi, Radhi Rafiee Afandi. A Survey of NewSQL DBMSs focusing on Taxonomy, Comparison and Open Issues. IJCSMC. December 2021. Volume 11, Issue 4.

[9] Tariq N. Khasawneh, Mahmoud Alsahlee, Ali Safieh. SQL, NewSQL, and NOSQL Databases: A Comparative Survey. In 2020 11th International Conference on Information and Communication Systems (ICICS)

[10] Mar á Murazzo1, Pablo G ómez, Nelson Rodr guez, Diego Medel. Database NewSQL Performance Evaluation for Big Data in the Public Cloud. In Book Communications in Computer and Information Science ((CCIS, volume 1050)), Naiouf, M., Chichizola, F., Rucci, E. (eds) Cloud Computing and Big Data. JCC&BD 2019.

[11] Andrew Pavlo, Matthew Aslett. What's Really New with NewSQL? SIGMOD Record. June 2016 (Vol. 45, No. 2).

[12] Maia, Francisco Carlos M. B. Oliveira. Sharding by Hash Partitioning A database scalability pattern to achieve evenly sharded database clusters. 17th ICEIS At: Barcelona, Spain, April 2015.

[13] A. Moniruzzaman. NewSQL: Towards Next-Generation Scalable RDBMS for Online Transaction Processing (OLTP) for Big Data Management. arXiv preprint arXiv: 1411.7343, 2014.

[14] Joachim Tankoano. Providing in RDBMSs the flexibility to Work with Various Non-Relational Data Models. Global Journal of Computer Science and Technology: H Information & Technology. 2023, Volume 23 Issue 2 Version 1.0. https://doi.org/10.34257/GJCSTHVOL23IS2PG1

[15] Chen, P., P-S. The entity-relationship model - toward a unified view of data. ACM TODS. March 1976, Volume 1, Issue 1. pp 9–36.

[16] Object Modeling Group. Unified Modeling Language Specification. October 2012, Version 2.5.

[17] Patrick Valduriez, Setrag Khoshajian, George Copeland. Implementation Techniques of Complex Objects. 12th Int. Conference on Very Large Data Bases - Kyoto, August 1986.

[18] Tirthankar LahiriSerge, Abiteboul Serge, Jennifer Widom. Ozone: Integrating Structured and Semistructured Data. 7th Int. Workshop on Database Programming Languages: Research Issues in Structured and Semi-structured Database Programming, December 1999.

[19] Marc H. Scholl. Extensions to the Relational Data Model. Available from: https://www.researchgate.net/publication/2381217_Extensions_to_the_Relational_Data_Model (accessed 29 March 2025).

[20] Joachim Tankoano. Mod èle relationnel imbriqu é. In SGBD relationnels – Tome 2, Vers les Bases de donn ées R éparties, Objet, Objet-relationnelles, XML, … Available from: https://www.researchgate.net/publication/366548683_SGBD_relationnels_-_Tome_2_Vers_les_Bases_de_donnees_Reparties_Objet_Objet-relationnelles_XML (accessed 29 March 2025).

[21] Z. Meral Ozsoyoglu, Li-Yan Yuan. On the normalization in Nested Relational Databases. LNCS. 1989, volume 361.

[22] Daniel J. Abadi, Samuel R. Madden, Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

[23] ORACLE. Oracle Database SQL Language. Reference 23ai, F47038-19, November 2024.

[24] Comer, D. The Ubiquitous B-Tree. Computing Surveys. June 1979, vol. 11, n °2.

[25] Valduriez P. Join Indices. ACM TODS. June 1987, Vol. 12, No. 2, Pages 218-246.

[26] Ashish P. Mohod, Manoj S. Chaudhari. Improve Query Performance Using Effective Materialized View Selection and Maintenance: A Survey. IJCSMC. April 2013, Vol. 2, Issue. 4, pg. 485 – 490.

[27] International Organization for Standardization (ISO). Information technology — Database languages SQL Part 16: Property Graph Queries (SQL/PGQ). (Edition 1, 2023), ISO/IEC 9075-16: 2023.

[28] Caio H. Costa, Jo ão Vianney B. M. Filho, Paulo Henrique M. Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, Xiaoli Zhou, Wenyuan Yu, Ying Zhang, Jingren Zhou. Towards a Converged Relational-Graph Optimization Framework. Proc. ACM Manag. Data, Vol. 2, No. 6 (SIGMOD), December 2024.

[29] Ronald Fagin, Phokion G. Kolaitis, Alan Nash. Towards a Theory of Schema-Mapping Optimization". PODS'08, June 9–12, 2008, Vancouver, BC, Canada.

[30] Jean B ézivin, Olivier Gerb é. Towards a precise definition of the OMG/MDA framework. Proc. 16th Annual Int. Conf. on Automated Software Engineering (ASE 2001)

[31] Chaimae Asaad and Karim Ba. NoSQL Databases: Seek for a Design Methodology. 8th Int. Conference, MEDI 2018, Marrakesh, Morocco, October 24–26, 2018.

[32] Carlyna Bondiombouy, Patrick Valduriez. Query Processing in Multistore Systems: an overview. [Research Report] RR-8890, INRIA Sophia Antipolis - M éditerran ée. 2016, pp. 38. hal-01289759v2.

[33] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform Access to Non-relational Database Systems: The SOS Platform. J. Ralyt´e et al. (Eds.): CAiSE 2012, LNCS 7328, pp. 160–174, 2012.

[34] Ágnes Vathy-Fogarassy, Tam ás Hugy ák. Uniform data access platform for SQL and NoSQL database systems. Information Systems. September 2017, Volume 69, Pages 93-105.

[35] Kwangchul Shin, Chulhyun Hwang, Hoekyung Jung. NoSQL Database Design Using UML Conceptual Data Model Based on Peter Chen's Framework. Int. Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 5 (2017) pp. 632-636.

[36] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, Gilles Zurfluh. Logical Unified Modeling For NoSQL DataBases. 19th ICEIS 2017, Apr 2017, Porto, Portugal. pp. 249-256. hal-01782574.

[37] Michael Stonebraker. NoSQL and Enterprises. Cacm | august 2011 | vol. 54 | no.