RESEARCH-ARTICLE

JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models

SHASHANK GUGNANI, Oracle Corporation, Austin, TX, United States

ZHENHUA LIU, Oracle Corporation, Austin, TX, United States

HUIJOE CHANG, Oracle Corporation, Austin, TX, United States

BEDA CHRISTOPH HAMMERSCHMIDT, Oracle Corporation, Austin, TX, United States

SRINIVAS KAREENHALLI, Oracle Corporation, Austin, TX, United States

KISHY KUMAR, Oracle Corporation, Austin, TX, United States

View all

Published: 22 June 2025

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

https://doi.org/10.1145/3722212.3724441
ISBN: 9798400715648

JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models

Shashank Gugnani
Zhen Hua Liu
Hui Chang
Beda Hammerschmidt
shashank.gugnani@oracle.com
zhen.liu@oracle.com
hui.x.zhang@oracle.com
beda.hammerschmidt@oracle.com
Oracle
Redwood City, California, USA

Srinivas Kareenhalli
Kishy Kumar
Tirthankar Lahiri
Ying Lu
srinivas.kareenhalli@oracle.com
kishy.kumar@oracle.com
tirthankar.lahiri@oracle.com
ying.lu@oracle.com
Oracle
Redwood City, California, USA

Douglas McMahon
Ajit Mylavarapu
Sukhada Pendse
Ananth Raghavan
doug.mcmahon@oracle.com
ajit.mylavarapu@oracle.com
sukhada.pendse@oracle.com
ananth.raghavan@oracle.com
Oracle
Redwood City, California, USA

Abstract

JSON Relational Duality is a novel solution to the object-relational impedance mismatch problem. By decoupling the data access model from the data storage model, application data can be read and written in terms of application-friendly hierarchical JSON documents using *Duality Views*, even when the underlying data is stored in the form of normalized relational tables (that can include JSON columns for schema flexibility). When data is stored natively as JSON documents, shared data is duplicated across many documents. With duality views, different JSON documents can efficiently share the same underlying data without any data duplication, or the risk of data divergence. Duality views also use a novel value-based optimistic concurrency control protocol that enables stateless APIs such as REST to be used across multiple reads and writes with full transactional consistency. JSON Relational Duality is a transformative breakthrough for application development that preserves the best of both worlds: Developer-friendly JSON-oriented access to application objects, combined with the power (efficiency, integrity, and composability) of the relational model.

This paper describes the concepts of JSON Relational Duality and various aspects of its implementation in Oracle Database Release 23ai. It also poses some interesting questions for further research.

CCS Concepts

• **Information systems → Data model extensions**.

Keywords

NoSQL, JSON, SQL, Object Relational Mismatch, ORM, Duality

ACM Reference Format:

Shashank Gugnani, Zhen Hua Liu, Hui Chang, Beda Hammerschmidt, Srinivas Kareenhalli, Kishy Kumar, Tirthankar Lahiri, Ying Lu, Douglas McMahon, Ajit Mylavarapu, Sukhada Pendse, and Ananth Raghavan. 2025. JSON Relational Duality: A Revolutionary Combination of Document, Object, and Relational Models. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724441

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724441

1 Introduction

The *Object-Relational Impedance Mismatch* problem is a long standing problem for application developers using relational databases. Applications prefer to process data in terms of use-case-specific hierarchical *objects*, while relational databases store and process data in terms of normalized *tables*. In a sense, pre-relational IBM-IMS [22] was closer to what today's developers want and JSON document databases can be viewed as IMS reincarnations.

Previous attempts to solve the object-relational impedance mismatch problem have had limited success. Some attempts [3, 5] stored hierarchical objects directly in the database which added to development complexity by introducing a complex object type system. Storing objects directly in the database also duplicates underlying data and loses the benefits of data normalization. In addition to object types, XML support was added to SQL [9], allowing the expression of hierarchical structures as XML. More recently, document databases [24] have tried to address this problem by storing application objects using a type-free text format, such as JSON. Support for JSON was added to relational databases [23], paving the way for a convenient data access model. However, it also introduced significant data duplication across documents[1] that share the same contents (such as different order documents that include the same product). Like any hierarchical data model, JSON suffers from the same problem as IMS: hierarchies need a root, and finding a single hierarchy that is suitable for all use cases is impossible for most applications. Therefore, data is stored in multiple hierarchies (JSON documents) and shared data needs to be duplicated.

Other attempts added Object Relational Mapping (ORM) layers [13, 25] to translate application object accesses to database table accesses, often introducing inefficient SQL and requiring multiple databases round trips for single object retrieval. Furthermore, ORMs are programming language-specific, making it impossible to share mapping information across microservices written in different languages.

[1]We use the words *document* and *object* interchangeably in this paper.

We propose a novel solution known as **JSON Relational Duality**, in which we **decouple the access model from the storage model**. Application object data is read and written in terms of application-friendly JSON documents, while the underlying storage may remain in normalized relational tables to eliminate storage duplication even if application objects have duplicated contents. With JSON Relational Duality, you can optionally include de-normalized storage (such as JSON datatype columns) for schema flexibility [18], obsoleting the need for document databases, and avoiding composition and de-composition cost for fully-owned objects. With JSON Relational Duality, JSON datatype-enabled relational databases can provide enriched document functionality that is superior to document databases, and far superior to the support and performance provided by glued-on ORM frameworks. We also introduce a novel optimistic concurrency control protocol to guarantee consistent updates to application objects, regardless of how much content duplication is present across the application objects. **JSON Relational Duality is a truly transformative breakthrough for application development that preserves the best of both worlds**: convenient and efficient document-based access to application objects; flexible, extensible, and highly efficient normalized relational storage along with schema flexibility and de-normalized document storage capability.

2 Related Work

The impedance mismatch between the flat relational model and the hierarchical object model has promoted research on updatable object views over relational storage. Literature [11] surveys two approaches to this. The first is a generic declarative approach based on the view definition. The second uses an abstract datatype approach with procedural coding. Keller et al. [15] follow the declarative approach that lays out the semantic foundational work on updatable object views over relational storage. However, since there is no RDBMS that supports a native hierarchical object data model inside a purely relational database, there is no implementation for either approach. The current popular industry practice for developers is to use Object Relational Mapping tools executed in the middle tier [13]. During the object-relational DBMS wave, both object-type views [2] and XMLType views [12] enabled users to construct hierarchical object views over the relational model in Object/XML-enabled RDBMS. However, updatable object-type and XMLType views where updates are automatically handled by the database are NOT fully supported by any database. Instead, users are required to write an instead-of-trigger [1] to update such object or XMLType views. The instead-of-trigger approach essentially follows the abstract datatype approach with procedural code discussed in [11]. On the other hand, XQuery (XQUF) updates on XML views over relational tables were researched in [10]. Separately, updates over XML views and their propagation to the underlying relational tables were implemented in [6] and [7]. Our approach is different from these works, as our DMLs are not just restricted to the propagation of updates of leaf scalars to updates of columns in the underlying tables. We also handle insertion and deletion of rows from the underlying tables. Our solution is comprehensive in nature since we also handle schema-flexible cases. Another key differentiation is that we compute both document- and row-level ETags for each input row so that updates can simply compare ETags instead of comparing every selected column value.

3 Key Contributions

To the best of our knowledge, JSON Relational Duality is the first practical solution that supports a DBMS native update of object views over the relational model in a completely declarative manner. Our solution leverages the popular JSON hierarchical data model support in a JSON datatype-enabled RDBMS [19]. It integrates the strength of both approaches outlined in [11] but also accomplishes the goal of native object view update fully inside DBMS as Keller et al. [15] describe. In addition, JSON Relational Duality provides many useful features, including optimistic concurrency control. The underlying tables composing a duality view can be updated concurrently (direct table updates). This is different than the approach proposed in [21] for updating XMLType objects that are shredded and stored in multiple relational tables that are internally owned and controlled by the XMLType object.

The key contributions of this paper are enumerated below.

(1) We introduce the concept of a 'Duality View' as a means of achieving document and relational duality. The duality view definition leverages and extends an intuitive GraphQL [30] like syntax to ease the hierarchical construction of JSON over a set of relational tables linked via primary key and foreign key references. The GraphQL syntax is conceptually like SQL macros over the standard SQL/JSON generation functions.
(2) The duality view definition allows declarative DML and concurrency control configurations using custom GraphQL directives. Thus, the approach proposed by Keller et al. [16] of using dialogs to control the DML translation between objects and tables is realized declaratively.
(3) The JSON object created from a duality view is a JSON datatype instance with OSON format [19] binary efficiency and built-in native schema flexible capability. OSON-based JSON datatype is as efficient as Snowflake's semi-structured datatypes [28] for supporting unstructured data. Duality views allow JSON datatype as a persistent column in addition to classical scalar datatype columns to achieve schema flexibility and de-normalization for performance at any hierarchical level. This means that our approach will enable data that does not conform to the underlying relational schema to be automatically stored inside a persistent native JSON datatype column. This is one key differentiator from [4] that requires tables to have a fixed schema that is fully normalized.
(4) The other key differentiation from [4] is that the JSON object instance constructed from duality view automatically computes and embeds an ETag and a system change number (SCN) as metadata inside the OSON [19] object. The ETag provides optimistic object update concurrency control and achieves object-update serializability via ETag validation pre- and post-object updates. The SCN provides the foundation for multi-version concurrency control with duality views. Applications can issue `AS OF SCN` queries to read consistent objects as of a certain timestamp or SCN. Although ORMs

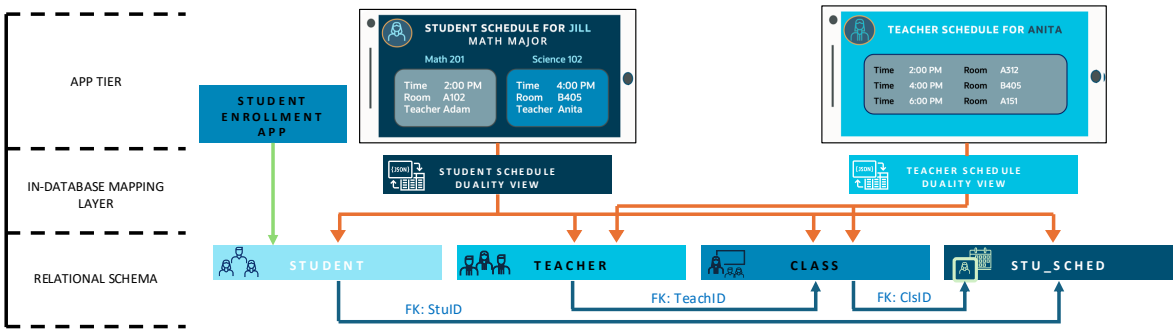


Figure 1: Overview of Duality Views (Green → Direct Table Access, Orange → Table Access via Duality Views, Blue → PK-FK Relationships)

can add a version column in a table to achieve lock-free updates at the row level, duality view ETag computation is hierarchically composable.

(5) Duality views provide query and DML parity with persistent JSON datatype-based popular NoSQL style JSON collections. Both SQL/JSON standard operators over JSON datatype and popular NoSQL APIs, such as MongoDB-compatible API and REST API, are supported over duality views. Queries over duality views are optimized using automatic rewrites of JSON path queries into relational queries leveraging SQL/JSON path algebra. The construction of duality view objects is optimized using top-down OSON template-based streaming generation. DMLs over duality views are optimized using object-level diffs to execute only the required modifications to the underlying tables. Moreover, application logic can be written on top of duality view objects in PL/SQL or JavaScript using a new declarative construct called directives.

4 Concepts

This section describes the key concepts of JSON Relational Duality.

4.1 Duality View

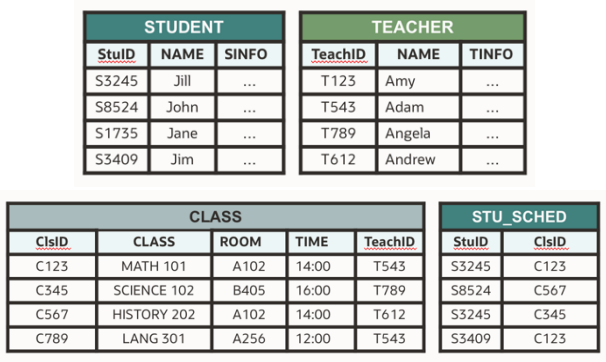
We introduce the concept of a 'duality view' as a means of achieving document and relational duality (i.e., document access and relational storage). A duality view exposes data stored in relational database tables as JSON documents. The documents are materialized – generated on demand, not stored as such. Duality views give data both a conceptual and an operational duality: it is organized both relationally and hierarchically. Different duality views can be based on data that is stored in one or more of the same tables, providing different JSON hierarchies over the same, shared data. A duality view directly defines and reflects the structure of JSON documents of a given kind (structure and field types). The JSON duality view is defined on top of a set of relational tables, which are conceptually linked via primary and foreign key joins to formulate a hierarchical view over the Entity/Relationship (E/R) graph. A duality view object is a sub-tree (directed acyclic graph) of the E/R graph with a single root table and (possibly many) dependent tables to formulate nested objects and arrays using the JSON data model. By default, tables can be joined automatically to realize documents of that kind based on the primary key (PK), unique key (UK), and referential constraint (RI) information recorded in the system catalog. Figure 1 shows a conceptual overview of duality views.

There are fundamentally two kinds of joins. **One-To-At-Most-One join** conceptually reflects the PK or UK of a table joined with the PK or UK of another table. **One-To-Many join** conceptually reflects the PK or UK of a table joined with the foreign key (FK) of another table. In the absence of such pre-defined PK/UK/FK information, users are required to declare them syntactically as part of the duality view definition. Note that the one-to-one and one-to-many linking relationships among entities is reflected as One-To-At-Most-One joins and One-To-Many joins, respectively. The many-to-many linking relationship among entities is reflected as a One-To-Many join followed by a One-To-At-Most-One join. Therefore, duality views can be used to support all common linking relationships among entities to formulate hierarchical objects.

Importantly, a duality view is **updatable**. So, a user may read a JSON document from the view, make any change to the document, and send it back to the database for update. The database automatically determines what has changed in the document and updates the underlying relational tables. The underlying update automation reflects and obeys the explicitly defined or implicitly inferred referential constraint relationships between the underlying tables and conforms to the updatability annotations declared on the duality view. A duality view must be able to translate a document update to row updates, so we do not allow arbitrary complex SQL query constructs (such as window functions or non equi joins) for updatable portions of a duality view. However, read-only parts of a duality view may use arbitrary SQL constructs.

4.2 Value-based Concurrency Control

Many real-world applications explicitly store versions numbers into relational tables to implement optimistic concurrency control manually in the application. This assumes a single hierarchy with the version number in the root row of the hierarchy. Every update to any row in the hierarchy must also update the version number in the root of the hierarchy. However, this needs to be implemented manually. We introduce the concept of value-based concurrency control as an alternative to the traditional lock- and version-based concurrency control methods. We use the concept of ETags to achieve this - an ETag represents a tag or hash value



STUDENT

StuID	NAME	SINFO
S3245	Jill	...
S8524	John	...
S1735	Jane	...
S3409	Jim	...

TEACHER

TeachID	NAME	TINFO
T123	Amy	...
T543	Adam	...
T789	Angela	...
T612	Andrew	...

CLASS

ClsID	CLASS	ROOM	TIME	TeachID
C123	MATH 101	A102	14:00	T543
C345	SCIENCE 102	B405	16:00	T789
C567	HISTORY 202	A102	14:00	T612
C789	LANG 301	A256	12:00	T543

STU_SCHED

StuID	ClsID
S3245	C123
S8524	C567
S3245	C345
S3409	C123

Figure 2: Sample Relational Schema for a Student Sched. App.

of the document contents. Each duality view document has an embedded ETag - which represents a collision-resistant hash of the document contents. ETags can be used for value-based optimistic concurrency control - an update to a duality view document is only accepted if the ETag of the document has not changed from a previous read. The advantages of using ETags for value-based concurrency control are twofold: (1) Updates to different duality views with overlapping data are automatically synchronized, and (2) no locks are required until the user wants to update a document. The advantage of value-based concurrency control over conventional lock-based concurrency control is twofold: (1) Concurrency control can be implemented across stateless APIs (such as REST) where transactions and locks cannot be held across database calls, and (2) it reduces the use of expensive locks commonly used in relational databases since locks do not need to be held during human or application think time.

5 Defining Duality Views

A JSON Relational Duality View is an updatable view that maps a hierarchical JSON object representation to a set of tables that are related by primary key and foreign key relationships. JSON Duality Views can be defined using both SQL and GraphQL, as described in sub-sections below.

5.1 Using SQL JSON for Defining Duality Views

Duality views can be defined using SQL/JSON generation functions. An object hierarchy can be obtained using nested scalar sub-queries. Data gathered from a sub-query is joined to data gathered from a parent table by a relationship between the corresponding columns in the sub-query's `WHERE` clause. Listing 1 shows an example of a duality view created using SQL/JSON syntax with multiple nested scalar sub-queries using the schema shown in Figure 2. In this case, the root table is student. For each student, the student table is joined with the stu_sched table using a one-to-many join to construct an array of schedule objects via a JSON array aggregation scalar sub-query. For each schedule, the schedule table is joined with the class table which in turn joins with the teacher table to pull columns of class and teacher tables and construct the schedule object. Note that the join from schedule to class and class to teacher is a one-to-at-most-one join, which is semantically enforced by a correlated

```
CREATE JSON RELATIONAL DUALITY VIEW
    Student_schedule AS
SELECT JSON {
  '_id'      : s.stuid,
  'student'  : s.name,
  'schedule' : [SELECT JSON {
    'scheduleId' : ss.scid,
    'class' : SELECT JSON {
      'classId' : c.clsid,
      'class'   : c.class,
      'time'    : c.time,
      'room'    : c.room,
      'teacher' : SELECT JSON {
        'teachId' : t.teachId,
        'teacher' : t.name
      } FROM teacher t WITH NOUPDATE
        WHERE t.teachid = c.teachid
    } FROM class c WITH NOUPDATE
      WHERE c.clsid = ss.clsid
  } FROM stu_sched ss WITH INSERT UPDATE DELETE
    WHERE s.stuid = ss.stuid]
} FROM student s WITH INSERT UPDATE DELETE;
```

Listing 1: Defining a Duality View using SQL

```
{
  "_id" : 8524,
  "student" : "John",
  "schedule" : [{
    "scheduleId" : 2,
    "class" : {
      "classId" : 567,
      "class" : "HISTORY 202",
      "time" : "14:00",
      "room" : "A102",
      "teacher" : {
        "teachId" : 612,
        "teacher" : "Andrew"
      }
    }
  }],
  "_metadata" :
  {
    "etag" : "1EFA13A3D6B5B81E97258F74B4F2CC1A",
    "asof" : "00000000000C20CE"
  }
}
```

Listing 2: Sample Student Schedule Document

scalar sub-query. Listing 2 shows a sample student schedule duality view document.

5.2 Using GraphQL for Defining Duality Views

Using the SQL/JSON generation function syntax can be complex, especially for document database users. Moreover, it is prone to bugs even for a seasoned SQL developer. We, therefore, conceptualize a fully functional GraphQL syntax as an intuitive and simpler way of creating a duality view. GraphQL syntax provides a significant advantage for duality views since it is easy to use and learn, and hides a lot of relational complexities that developers working with object or document models are not very familiar with.

```
CREATE JSON RELATIONAL DUALITY VIEW
    Student_schedule AS
student @insert @update @delete {
  _id : stuid
  student : name
  schedule : stu_sched @insert @update @delete [{
    scheduleId : scid
    class @noupdate {
      classId : clsid
      class
      time
      room
      teacher @noupdate {
        teachId
        teacher : name
      }
    }
  }]
};
```

Listing 3: Defining a Duality View using GraphQL

GraphQL [30] is an open-source query and manipulation language for APIs. As a language, it was first developed at Facebook in 2012 and open-sourced in 2015. Subsequently, the GraphQL project has moved under the Linux foundation. GraphQL defines the output of a query in a JSON-document-like hierarchical structure. In other words, GraphQL provides a what-you-see-is-what-you-get (WYSIWYG) syntax for creating duality views. Listing 3 shows an example of a duality view created using the GraphQL syntax. Notice that the DDL statement itself defines the output of a query in a JSON-like structure. The fields in this JSON object are strongly typed objects that can be linked together like a graph in a GraphQL schema. The GraphQL query syntax can thus be used as a shorthand for duality view definition without explicit specification of nested scalar subqueries. The methods to automatically derive a GraphQL schema from underlying relational objects and to construct a duality view from the schema are described in subsequent sections.

Another useful feature of GraphQL is the notion of 'custom directives' which can be used to customize the output of a GraphQL query. For instance, a directive can be used to unnest attributes of a child object into its parent object. We use custom directives wherever the GraphQL standard does not have an equivalent today. Some of these directives may be more generally applicable and can be standardized.

5.3 Deriving a GraphQL Schema

A GraphQL schema represents a graph of strongly typed objects. Tables in a relational schema can be considered vertices in a graph, with FK relationships representing the edges. Thus, our sample relational schema can be represented as the graph shown in Figure 3.

Converting this graph to a GraphQL schema involves following a simple set of rules listed below.

(1) Every table T is mapped to a GraphQL object type. The name of the object type is the same as the table name, except that the first character is capitalized and the remaining characters are in lowercase. E.g., the object type for a table named STUDENT is `Student`.

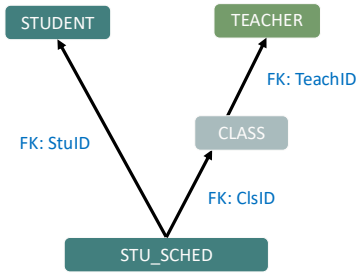


Figure 3: Student Schedule Relational Schema as a Graph

```
type Student {
  stuid: String!
  name: String
  sinfo: String
  stu_sched: [Stu_sched]
}

type Teacher {
  teachid: String!
  name: String
  tinfo: String
  class: [Class]
}

type Class {
  clsid: String!
  class: String
  room: String
  time: String
  teacher: Teacher
  stu_sched: [Stu_sched]
}

type Stu_sched {
  clsid: String
  stuid: String
}
```

Figure 4: Student Schedule GraphQL Schema

(2) An object type T's field definitions are based on the columns in table T's definition. Each scalar column 'f' of table 'T' is mapped to a scalar field of the same name as f but in lowercase. E.g., column CLSID of type VARCHAR2 is mapped to a GraphQL field clsid of type String. If a column has a NOT NULL constraint, it is marked as a non-null field in the schema.

(3) If a link T->T1 exists via a foreign key 'fk', then we add an object field (an object field is a field with type Object) in the object type T with the field name as the object type name corresponding to the parent table T1. The field type is the object type name corresponding to the parent table 'T'. E.g., for a foreign key column TEACHID in table CLASS which references table TEACHER, an object field teacher of type Teacher is added to the Class object.

(4) If a link T2->T exists, via a foreign key 'fk' on the child table T2, then we add an object field 'fk' to the object type T where the field name is the same as the object type name corresponding to T2 (in lowercase). The return type of the field fk is an array of the object type corresponding to T2. E.g., when CLSID is a foreign key from table STU_SCHED to CLASS, we add a field stu_sched of type [Stu_sched] to the object Class.

(5) GraphQL's built-in scalar types include String, Float, Int, Boolean, and ID. We add the following custom scalar types to allow better representation of Oracle native data types: Date, Timestamp, Binary, Vector, and JSON.

Based on this set of rules, the GraphQL schema corresponding to the STUDENT schema is shown in Figure 4.

5.4 Defining Duality Views using a GraphQL Schema

Deriving a GraphQL schema corresponding to a relational schema allows us to construct the expected query output from the duality view in its GraphQL syntax. Defining a duality view on top of this query is then a simple extension of the SQL syntax.

Consider the schema from Figure 2 and the STUDENT_SCHEDULE duality view from Listing 1. The GraphQL equivalent DDL representing this duality view can be derived as follows.

(1) Start by listing the tables included in the duality view and add the corresponding GraphQL objects to the expected query output. Remember to start from the 'root' object or the table on which the view is created.
(2) For each related sub-object required, add a level of hierarchy into the expected query output.
(3) Include fields at each object level based on whether they need to be a part of the duality view. Note that fields might be of type JSON array or object depending on the parent-child relationships in the relational schema.
(4) Prefix the GraphQL query with '`CREATE JSON RELATIONAL DUALITY VIEW <view_name> AS`'.

5.4.1 Custom Directives For Ambiguous Cases. In many cases, ambiguities may arise in the GraphQL DDL that require clarification using declarative annotations (directives). For example, it is possible to have multiple foreign key relationships between two tables, in which case, the expected query output needs to specify which foreign key will be used to perform the join or fetch. Similarly, a self-join might result in a cyclic definition if the correct foreign key column is not specified. We use custom GraphQL directives to achieve this. The GraphQL directive `@link(from: [String1], to: [String2])` allows us to specify the foreign key to be used for joining two types (tables). The 'from' argument indicates that the foreign key column name represented by 'String1' needs to be used to join from this table to its parent table, whereas the 'to' argument is used to indicate that child rows fetched should correspond to the column specified by 'String2'. In the case of a self-join, the parent and child tables are the same. For multi-column FKs, an ordered array of column names is passed to the to and from arguments.

5.4.2 Re-shaping Output. Nesting refers to grouping several attributes of an object as a sub-object of that object, while unnesting refers to externalizing parts of a sub-object into its parent object. The GraphQL standard does not allow the specification of nesting or unnesting in a query. We, therefore, support nesting and unnesting through extension to the standard, as follows.

(1) We add a self-association attribute in the GraphQL object schema. E.g., the Student object contains a field of type Student.
(2) We introduce custom directives called `@nest` and `@unnest` to achieve the effect of nesting and unnesting. E.g., Unnesting simply requires addition of an `@unnest` directive to the field(s) or sub-object(s) that need to be unnested into the parent object.

5.4.3 Updatability Annotations. Duality views are updatable using both SQL and JSON syntax. Moreover, the duality view definition lets developers specify whether each field or each sub-object in the view is updatable. Such restrictions are especially important to achieve different business use-cases, for example, prevent updates to dimension tables which are rarely modified. For instance, the Teacher sub-object cannot be modified from the Student_schedule view, but it can be modified from the Teacher_schedule view. We support the following directives to specify updatability for each SQL operation. The duality view updatability related annotations include `INSERT` / `NOINSERT`, `DELETE` / `NODELETE`, and `UPDATE` / `NOUPDATE`. These annotations specify whether users can perform INSERT, UPDATE, or DELETE operations on a specific sub-object (table). In addition to these, we also support two additional annotations to specify whether the value for a field (column) or sub-object (table) should be included in the ETag for the duality view object: `CHECK` / `NOCHECK`. All of these annotations are expressed in GraphQL using custom directives: `@insert` / `@noinsert`, `@delete` / `@nodelete`, `@update` / `@noupdate`, and `@check` / `@nocheck`.

5.4.4 Predicates for Filtering. By default, the `WHERE` clause in duality views is used to define join key columns. However, users are allowed to add additional filter predicates on base table columns. Semantically, filter predicates behave just like predicates on regular views with the check option. Predicates are processed as additional check constraints when performing DMLs over the underlying relational tables.

5.4.5 Flex Clause for Schema Flexibility. Duality views provide full schema flexibility [8, 18] like pure JSON databases. A column of data type JSON holds the schema flexible content and allows two modes: If the JSON datatype column is explicitly referenced, then the flexible object content has an explicit key associated with it. Otherwise, the JSON datatype column is implicitly used to support fields that are not mapped to predefined columns of a table. In that sense, it serves as an 'overflow' column. When constructing the document, the content of the flex column is merged into the corresponding JSON object such that a user cannot tell which parts are mapped to columns versus using the flex column.

6 Access APIs for Duality Views

Duality views expose data as JSON documents for read and write access. For simple single document access, each document has a unique ID (usually mapped to primary key columns) which is also indexed for fast retrieval. In addition to this key/value style access, it is also possible to search for documents using filter criteria or even run aggregation operations on top of multiple documents. Multiple APIs are supported to allow different developer personalities to interact with the data in their preferred way.

6.1 SQL

The SQL standard has introduced a dedicated JSON datatype and several operators to work with JSON data [23]. For example, `JSON_VALUE` and `JSON_QUERY` to extract parts of JSON data, `JSON_EXISTS` to select data based on filter predicates, and `JSON_TABLE` to map the hierarchical JSON data to flat rows and columns. All operators use path expressions to navigate inside the JSON data to

select one or multiple values. For example, the path expression `$.address[*].zip` navigates to all values of the field `zip` under `address`. The path step `[*]` first selects (or iterates over) all items of the `address` array.

Duality views expose JSON data in a single column using JSON data type. The column is always called `DATA`, therefore clients can issue database operations without an extra round trip to first find out column names and types. The view can be queried using any SQL/JSON operator. In many cases, the path expression of the SQL/JSON operator matches to a column of an underlying table. In that case, the query gets optimized and yields full relational performance without any JSON processing overhead. Duality views are fully updatable and, therefore behave just like tables. Updates can be full document replacements or piecewise updates using the `JSON_TRANSFORM` or `JSON_MERGEPATCH` operators.

6.2 REST API

Using the Oracle REST Data Services (ORDS) technology [26], duality views can be 'REST-enabled'. This automatically creates a complete set of REST APIs (endpoints) to work with data: standard HTTP commands like PUT, GET, PATCH, or POST provide functionalities to load, update, or fetch JSON documents. Each HTTP operation is automatically translated to an equivalent SQL operation by ORDS. Therefore, the database can pick the best execution plan and leverage indexes automatically. For existing (legacy) relational applications, the use of duality views and REST is an easy way to implement the microservice architecture without changing the application.

6.3 MongoDB-Compatible API

MongoDB [24] is a NoSQL database that organizes JSON documents in collections. In addition to key/value semantics, MongoDB provides document filters, JSON transformations, transactions, and limited analytical capabilities. An additional API provides MongoDB compatibility for duality views: each view is automatically also a MongoDB-compatible collection. Documents can be easily identified by the '_id' field, which maps to the primary key columns of the root table of the duality view. Similar to the REST API that translates HTTP commands to equivalent SQL, the 'Oracle Database API for MongoDB' also translates MongoDB wire-protocol commands to SQL. For example, a MongoDB filter expression like `Student_schedule.find({"student" : "Jill"})` gets translated to the equivalent SQL/JSON statement:

```
SELECT sch.data
FROM Student_schedule sch
WHERE JSON_EXISTS(sch.data, '$.student?(@==:b1)'
      PASSING 'Jill' as "b1")
```

This intermediate query then gets further optimized by the database to pick the right column from the 'student' table:

```
SELECT sch.data
FROM Student_schedule sch, student stu
WHERE stu.STUDENT = :b1
```

This relational query then gets planned by the optimizer to use indexes and materialized views, when possible.

All these query rewrites and optimizations are transparent to the user using path query rewrites over relational storage [17]. With the 'Oracle Database API for MongoDB', it is possible to connect MongoDB tools, drivers, and frameworks directly to duality views. A MongoDB user would not need to know how the documents are being stored. This is a major benefit for document API (and REST) users: the duality view abstracts the storage from the access, allowing users to redesign the storage without affecting clients as long as the JSON shape remains the same.

6.4 JSON Schema

An application programmer who works with JSON documents returned by duality views can also use a JSON schema description. It is auto-generated from the underlying relational schema and describes the object's shape, data types, constraints, and keys as a standard JSON schema document. Schema flexible content can be further described by a JSON dataguide, a JSON schema derived from instance data. We have extended the JSON schema standard with a database-specific vocabulary to capture the richer SQL type system and other database-specific information.

7 Queries on Duality Views

This section covers the JSON and ETag generation process when a duality view is queried. It also describes how queries are optimized using automatic rewrites.

7.1 JSON Generation

The duality view definition specified via GraphQL is transformed into SQL/JSON generation functions to assemble relational data into a JSON object as if the view were defined using the full SQL/JSON generation syntax as shown in Listing 1. `JSON_OBJECT()` is used to construct a JSON object from columns of one table or a set of tables linked via one-to-at-most-one join without row cardinality expansion. `JSON_ARRAYAGG()`, on the other hand, is used to construct a JSON array by assembling a set of rows linked via one-to-many join with row cardinality expansion. A one-to-at-most-one join JSON object output can be UNNESTED into the parent object, if needed. It is internally evaluated as a correlated scalar sub-query to enforce one-to-at-most-one join semantics which raises a runtime error if the join has more than one matching row. When there is no matching row, the correlated scalar sub-query semantically behaves like a left outer join. The un-matching row content is returned as either an empty JSON object for the NEST-ed object construction option or as the absence of an object for the UNNEST-ed object construction option.

We optimize the object construction query by leveraging top-down streaming evaluation of the SQL/JSON generation operator tree. If we evaluate in a bottom-up manner, we will generate many intermediate JSON objects, which is very inefficient. Rather, by using the top-down approach, one final JSON object is created by the top-level `JSON_OBJECT()` operator. All the nested `JSON_OBJECT()` and `JSON_ARRAYAGG()` operators assemble data into the final output JSON object directly. For single-table duality views, we internally optimize the `JSON_OBJECT()` construction even further using an OSON template that reflects the underlying table columns with a

Algorithm 1 ETag Computation Algorithm

```
1:  function EtagJsonObj(json_obj_expr joe)
2:      etag ← init_seed
3:      for all sorted scalar fields sf in joe do
4:          etag ← ChainHashFunc(etag, GetValue(sf))
5:      for all sorted json objects jo in joe do
6:          etag ← ChainHashFunc(etag, EtagJsonObj(jo))
7:      for all sorted json arrays ja in joe do
8:          etag ← ChainHashFunc(etag, EtagJsonArr(ja))
9:      return etag
10: function EtagJsonArr(json_arr_expr jae)
11:     etag ← init_seed
12:     for all sorted json objects jo in jae do
13:         etag ← Xor(etag, EtagJsonObj(jo))
14:     return etag
```

fast OSON construction phase. The idea is to create a binary template (just like an online form) that already contains the structure and type information for fields in the object. When a document in the view is read, the pre-created template is loaded and filled with column values read from the table.

7.2 ETag Generation

ETag computation is a chained process reflecting the document construction order. ETag checked field values are fed into a hash function based on lexicographical field name order. Similarly, the ETag from each child object is fed into the hash function based on lexicographical object field name order. For arrays, ETags of objects comprising the array are XORed to create an aggregate ETag. Algorithm 1 describes the pseudocode for ETag computation. Note that each row must have a unique/primary key that is part of the row ETag to ensure that the XOR result does not cancel values from different rows.

The root JSON object contains the ETag value for the object. To avoid computing the ETag in the post-evaluation path after JSON object construction, we compute the ETag in the same evaluation process as the SQL/JSON generation operator tree evaluation. Each `JSON_OBJECT()` and `JSON_ARRAYAGG()` operator has metadata to determine which column values are required to compute the ETag and uses them to compute the final ETag value while assembling the JSON object. The `JSON_OBJECT()` operator computes the ETag for a row using its column values. The `JSON_ARRAYAGG()` operator computes the ETag for a set of rows as an aggregate value from ETag values computed from each input row. During root JSON object evaluation, the final ETag value computed from the top-down process is returned and added to the final JSON object output of the duality view.

7.3 Optimization of Queries

`JSON_VALUE()` and `JSON_EXISTS()` queries over duality views go through a query rewrite phase which converts the SQL/JSON path predicates into equivalent relational predicates. This allows such operations to be pushed down to the underlying relational tables and avoids JSON object construction whenever possible. Query rewrite uses query algebra reduction rules on the `JSON_VALUE()` and `JSON_EXISTS()` operators over `JSON_OBJECT()` and `JSON_ARRAYAGG()` operators, and is conceptually similar to query rewrites over XML [17].

8 Updates to Duality Views

Duality Views allow users to access relational data as hierarchical JSON documents or objects. Access APIs support reads or GETs of objects and modifications of the object itself (PUTs). Duality views can be modified using SQL syntax, Oracle JSON syntax, and REST APIs. This section expands on the internals of updating a duality view object.

A duality view is a view built on top of tables related by PK and FK relationships, and an object in this view corresponds to a set of rows related by PK and FK constraints. Modification of fields in the JSON object therefore results in the modification of rows in the underlying tables that construct this object. Consider an object in the Student_schedule duality view. A valid insertion (or POST) of a Student_schedule object in this view will result in the addition of a new student row in the STUDENT table and insertion of one or more rows in the STU_SCHED table based on the classes this student intends to take. An invalid insertion, on the other hand, would result in an error. For example, if the StuID already exists, or if the classes listed in the new object are not part of the CLASS table, then it is considered an invalid insert. Decomposition of an object-level modification into DMLs on the underlying tables and validation of the operations therein are performed by three specialized modules in the duality view infrastructure: Shredder, Diff-Computer, and Row-Updater.

In addition to basic modifications, duality views allow view definitions to specify PK-FK and custom constraints on the updatability of underlying tables. This is especially important in the case of dimension tables. For example, the insertion of a new Student_schedule object cannot result in the addition of new classes, i.e. rows in the CLASS table. This is a custom rule that needs to be specified during the creation of the duality view using additional SQL clauses as discussed in section 4.4. The enforcement of such custom rules is done by the Row-Updater module.

8.1 Shredder

The duality view DDL driver analyzes the entire query in the view definition to enforce required semantics and constructs a metadata JSON document that captures the duality view definition. In particular, it analyzes linking relationships among tables that are used in the view definition and then creates a **duality view metadata tree** as a single JSON document that captures all the involved tables' metadata.

During the production of a JSON document for a particular duality view, the SQL/JSON operator tree is traversed and a JSON document is written without concern for duplication. Shredding the object back into rows is more complex, and requires the duality view metadata tree to drive the process. The tree contains a cross-reference that traces fields back to the row and column of origin. When a document is received, it is decomposed by a depth-first traversal that creates rows and populates the column values from the fields originating from them.

Rows are placed into hash tables and explicitly linked to their immediate parent row on the tree. The hash tables are keyed by the primary key, and this process allows duplicate rows within a table instance to be resolved. Resolution requires that all column values match; if not, it is considered an error.

During the process of disassembling a document back into rows, some table instances may offer support for flexible fields. That is, fields in the document that are not bound to any specific column of origin can be stored in a JSON object specifically provided for them. If a table offers flex-field support, fields that are not traceable to a column or a calculation are assembled into a new value for the flexible column.

8.2 Diff-Computer

Update operations against a duality view can be done either by replacement of a document with a new version, or by sending a change set (a difference or patch document).

The replacement process works by first running the document through the disassembly process to re-create the tree of rows and resolve any duplicates. Both the new replacement version (proposed by an application) and the original version (as read from the database) undergo this process, although as an optimization the original version can be directly created as a set of rows and bypass the disassembly process.

Once both documents exist as hash tables with row instances, the rows are compared by a 'differencing' engine to classify them as unchanged, new, removed, or changed. Only the new, removed, and changed rows are reported to the Row-Updater. This reporting is done in a pre-determined visitation order. Changed rows include information about exactly which columns have changed. The Row-Updater can determine if a new or removed row implies a need to insert or delete a row in the underlying table.

8.3 Row-Updater

The Row-Updater module is responsible for updating the tables underlying the duality view based on the 'diff' calculated by the Diff-Computer. It relies on constraints defined on tables and updatability annotations defined on the duality view to understand user intent behind a specific DML. It guarantees no-harm updates to tables - (1) on an update to a duality view document, only rows that are part of the document are modified, (2) rows are unlinked instead of deleted whenever possible, and (3) rows in read-only tables are never modified. We run SQL statements to update the underlying tables, so all integrity constraints defined on the underlying relational tables are enforced as usual.

For understanding user intent, this module relies on the following rules:

(1) UPDATE: A row is updated if it is marked updated or inserted by the Diff-Computer, it already exists in the database (based on key lookup), and the table containing the row is annotated as updatable in the view definition. It may also be updated if it is marked deleted by the Diff-Computer, and the table containing the row is annotated as updatable but not deletable. In this case, the row is simply unlinked from its parent row.

Algorithm 2 Isolation Algorithm

```
1:  function Update(usr_obj, usr_etag, _id)
2:      cur_obj, cur_etag ← ReadObj(_id)
3:      if cur_etag ≠ usr_etag then
4:          throw error
5:      cur_rowset ← Shredder(cur_obj)
6:      usr_rowset ← Shredder(usr_obj)
7:      diff ← DiffComputer(usr_rowset, cur_rowset)
8:      RowUpdater(diff)
9:      usr_etag_new ← ComputeETag(usr_obj)
10:     cur_etag_new ← GetETag(ReadObj(_id))
11:     if cur_etag_new ≠ usr_etag_new then
12:         throw error
```

(2) INSERT: A row is inserted if it is marked inserted by the Diff-Computer, it doesn't already exist in the database (based on key lookup), and the table containing the row is annotated as insertable in the view definition.
(3) DELETE: A row is deleted if it is marked deleted by the Diff-Computer, the table is not shared (dimension), and the table containing the row is annotated as deletable in the view definition. Shared rows are simply unlinked from any dependent rows.

Based on the rules described above, the row updater generates SQL statements to update the relational tables based on the object diff obtained from the diff computer. Importantly, it only generates statements to update rows and columns that have been modified by the user (values not modified by the user are left untouched). This enables fast update operations on duality views.

8.4 Isolation and Concurrency Control

Providing ACID transactions on duality views is a must for a practical solution. By leveraging the existing transactional properties of a relational database, atomicity, consistency, and durability of transactions are guaranteed out-of-the-box. That is because a duality view DML gets broken up into several row DMLs that all execute in the same transaction. Isolation, however, is non-trivial to support. Since data is stored as relational rows, a single object can comprise multiple rows. Updating a single document may involve updating multiple rows in discrete steps which may be interleaved with concurrent updates to the same rows by other sessions. This may cause anomalies such as lost writes (e.g., an update incorrectly overwrites a field updated by a concurrent update to its previous value) or inconsistent updates (e.g., two sessions are able to concurrently modify the same object).

An obvious solution to this problem is to lock all rows comprising the object before modifying it. This approach has several performance issues. First, locking reduces concurrency. Since a duality view need not include all columns in a table, locking an entire object would unnecessarily prevent modifications to columns not part of the view. Second, locking in relational databases may be a disk-based change that requires the generation of redo. For large objects comprising several rows, this can be quite expensive.

To solve this problem, we reuse the client-side value-based concurrency control scheme for server-side isolation (or concurrency

control). Algorithm 2 provides the pseudo-code for our approach. The algorithm consists of three parts, pre-validation (lines 2-4), data-update (lines 5-8), and post-validation (lines 9-12). During pre-validation, we compare the ETag contained in the user object with the ETag of the current version of the document in the database. This step allows users to realize client-side concurrency control and guarantees that there are no lost writes. However, it is not critical for achieving isolation of object updates within the database. During data-update, we shred both the old (cur_obj) and new (usr_obj) objects into a set of rows. We then diff these row sets to find out what has changed in the object. Finally, we update the relational tables based on the diff. Note that rows could have been concurrently updated between the time we read the object containing the rows (line 2) and we update the rows (line 8). To guard against such consistency issues, the SQL statements used to update rows contain a predicate on the row ETag to ensure that the row has not been modified since the object was read (e.g, `UPDATE student SET name = :name WHERE stuid = :id and ROW_ETAG = :cur_row_etag`, where `ROW_ETAG` is a virtual column representing the ETag of the row). As rows are updated, row locks are automatically taken by the database. During post-validation, we compare the expected ETag of the user object (after update) with the ETag of the current version of the document in the database. This ensures that any rows in the object that were not modified by the user are also not modified by a concurrent session. Since it is possible for a concurrent session to have modified the unchanged rows in the object and still not have committed the transaction, the computation of the current version of the ETag (in line 10) must also read uncommitted changes from concurrent sessions. Once post-validation is complete, we can safely assume that two transactions did not concurrently modify the same object through the same duality view and that the update to the object appeared to happen as of a single point in time.

The proposed isolation algorithm is an optimistic concurrency control algorithm, so updates on duality views may fail when a conflict is detected, and the user must retry the update later. In highly concurrent environments (which we expect to be rare), this algorithm may result in frequent conflicts and repeated retries. We have designed two approaches to handle these environments. First, users can annotate fields or objects as `NOCHECK` to exclude them from ETag computation. This prevents conflicting concurrent updates on those fields or objects from failing and increases transaction throughput. Second, users can lock an entire object using the `SELECT FOR UPDATE` statement on a duality view. This enables traditional lock-based concurrency control on duality views. Users can thus choose the appropriate concurrency control solution for their use-case based on workload concurrency requirements.

9 Directives on Duality Views

Duality views allow applications to represent application objects on normalized relational data along with SQL/REST operations. In addition, applications usually need to extend these operations on application objects with additional logic. Some additional supplemental logic that may be needed in a typical database interaction includes:

```
1  CREATE DIRECTIVE AddCourseCnt FOR Student_schedule
2  AUGMENT ON SELECT
3  USING
4  DECLARE
5    jsonobj json_object_t;
6  BEGIN
7    jsonobj := json_object_t(:OLD.data);
8    jsonobj.put('courseCount',
9                :OLD.data.schedule[*].count());
10   :NEW.data := jsonobj.to_json;
11 END;
```

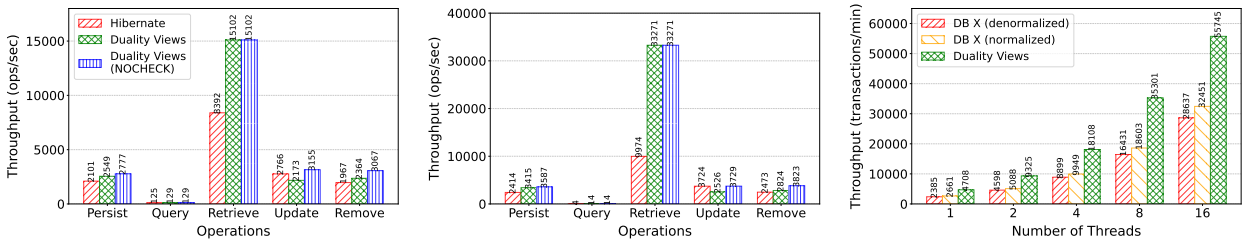
Listing 4: Defining a Directive on a Duality View

(1) Augmentation: Augment the object with additional values on a read or a modification. E.g., Compute the total course count or GPA for a student.
(2) Validation: Ensure that an object (including its augmented values) conforms to business rules. E.g., Limit the maximum number of courses a student can register for in a semester.
(3) Notification: Generate a notification that an object has been created, modified, or deleted via a history table, an event queue, or even via the redo log. Notification has many uses such as history tracking, replication, and event processing. Regardless of how an object is changed (direct update to tables or updates to duality views with overlapping content), we track all affected objects. E.g., When the timings for a class change, notify all registered students of the change.
(4) Continuation: An action on an object may require a follow-up action. Continuations are needed to facilitate workflow-style processing. E.g., When a new class is added, send an email to facilities to schedule the time and room for the class.

Directives are a new declarative construct that extend the capabilities of duality views to allow the specification of supplemental logic for duality view operations. They have the following properties:

(1) Deterministic: The outcome of each directive is deterministic and transactional, based only on the sequence of the operations within the directive.
(2) Declarative: Directives are defined via declarative SQL constructs (that may involve expressions or side-effect-free functions). Only continuations, which are downstream operations, are allowed to use arbitrary procedures.
(3) Use-Case Oriented: Directives are use-case specific, i.e. different directives can be used on different use-case-specific views. New use cases can use new views or directives but should not require changes to the underlying table declarations.
(4) Tunable: For performance, an application can choose to apply the directive only to the object(s) explicitly being modified. For completeness, an application can also choose to have the directive apply to the closure of all modified objects.

Importantly, directives enable applications to add business logic directly on objects instead of rows, simplifying the development process. Directives on duality views are supported using new SQL syntax. The procedural logic for a directive can be specified in



(a) JPAB Throughput (Batch Size = 5) (b) JPAB Throughput (Batch Size = 5000) (c) TPC-C-Duality Throughput (Scale Factor = 100)

Figure 5: Performance Comparison with Hibernate and DB X

PL/SQL or JavaScript. Listing 4 shows the DDL for creating an augmentation directive on a duality view that adds a `courseCount` computed field to the output object on a `SELECT`. The value of the field is computed using a JSON path expression. `:OLD.data` represents the input object and `:NEW.data` represents the output object.

10 Performance of Duality Views

This section discusses performance aspects of duality views compared with ORMs and document databases. Specifically, we compare duality views with Hibernate [27], a popular open-source ORM, and DB X, a popular open-source document database. Our goal is to verify through thorough experimental evaluation that duality views provide performance that delivers the best of both ORMs and document databases.

10.1 Performance Testbed

Our testbed consists of an Oracle Exadata cluster (Exadata X8M-8 HC). Each server in the cluster has eight Intel Cascade Lake CPUs (8268@2.90GHz), 3TB DRAM, and 192 cores. The client and server are run on separate servers with 14 additional servers used for storage. We use Oracle database release 23ai, Hibernate version 5.2.11, and JDK 11 for all tests.

10.2 Comparison with Hibernate

We use the JPAB benchmark [14] collection test to compare with Hibernate. The schema contains one entity class that represents a person. The Person class contains an auto-generated integer primary key field, nine string fields, one integer field, three date fields, and a `List<String>` collection field containing 1-3 phone numbers. The test executes the following 5 CRUD operations:

(1) Persist: Person objects with randomized data are inserted into the database.
(2) Query: Person objects are queried from the database using a predicate on a string field (last name starts with a random prefix).
(3) Retrieve: Objects are retrieved from the database by using the primary key (random primary key values are selected for retrieval).
(4) Update: Random person objects are retrieved from the database (by their primary key), updated in memory, and then sent to the database.
(5) Remove: Random person objects are retrieved from the database (by their primary key) and then deleted from the database.

We simulate two batch sizes, small and large. In the small mode, transactions are committed after every 5 operations, and objects are retrieved in groups of 5. In the large mode, transactions are committed after every 5,000 operations, and objects are retrieved in groups of 5,000. We compare operation throughput with duality views in two modes. The first is DEFAULT, where all fields in the duality view are checked and included in the ETag. The second is NOCHECK, where all fields are excluded from ETag computation, effectively resulting in the isolation algorithm being skipped. NOCHECK represents an apples-to-apples comparison with Hibernate since it does not provide server-side isolation of DMLs.

Figures 5(a) and 5(b) show the results of this experiment. In the DEFAULT mode, duality views outperform Hibernate in both modes for all operations, except update. By building the mapping interface into the database, duality views avoid multiple round-trips to resolve a single operation. Further, queries are optimized by pushing operations to the relational tables and DMLs are optimized by only modifying columns mapping to fields modified by users. For updates, duality views are slower due to the overhead of the isolation algorithm. When the isolation algorithm is disabled (NOCHECK mode), duality views outperform Hibernate in all operations. Since Hibernate does not offer server-side isolation of updates, applications typically need to explicitly handle concurrency control, for example, by adding a version field to objects. This approach reduces operation throughput and is generally not scalable. Therefore, we concluded that duality views offer superior performance compared with ORMs (up to 3x better query throughput and up to 40% better DML throughput) despite providing stronger isolation and concurrency control guarantees.

10.3 Comparison with Document DB X

We use an extended version of the TPC-C [31] benchmark, called TPC-C-Duality, to compare with DB X. We use a denormalized schema as well as DB X's recommended TPC-C schema – collections are normalized and mirror the relational schema used by duality views (except order lines are embedded inside order documents)[2]. Figure 5(c) shows the results of this experiment. Duality views outperform DB X by up to 2x on transactions per minute

[2]This schema was shown to perform the best for TPC-C.

across all concurrency levels. The main reason for this is that DB X (normalized) requires multiple round-trips to the database for every transaction since multiple collections need to be updated for every transaction. With the denormalized schema, each transaction can be processed in a single round-trip. However, there is significant data duplication across documents. A single transaction needs to update many documents and overall performance is worse compared with the normalized schema design. On the other hand, one duality view is created for each TPC-C use case, without any data duplication. A single transaction requires an update to exactly one duality view with a single round-trip to the database.

11 Future Work

This section discusses future research avenues for JSON Relational Duality.

11.1 Schema Evolution

The relational model often assumes a perfect schema to begin with and is confronted with schema evolution issues as application business requirements evolve. The JSON datatype has complemented the relational model by partially solving the schema bootstrapping and evolution problem to allow high entropy data to be stored, queried, and indexed without up-front schema design. Such high entropy data should be self-contained without the need to have a centralized schema definition [18]. A data guide [20] over high entropy JSON datatype instances can be obtained so that schema-based relational access can be defined as `JSON_TABLE()` views over such JSON data. Therefore, the physical storage of duality views can be either a set of schema rigid tables, JSON datatype-based document collections containing schema flexible high entropy data, or a combination of both. Duality views form a layer of abstraction or indirection over the underlying physical schema, allowing users to evolve the physical schema while keeping the same logical schema. Duality views offer better schema evolution than both relational and document databases, since new views can be created for new use cases without impacting existing use cases. To provide seamless schema and application evolution, we need extensive research on two fronts. First, there is a need for future work on automatically defining and evolving the duality view definition by *analyzing the entropy of data*. Second, there is a need for future work on identifying the changes required to the layer of abstraction (duality views), such that *the physical schema can be changed in any way without impacting the data access APIs*.

11.2 Normalization

Relational normalization avoids data duplication and data update consistency issues by storing common data only once. Proper use of JSON datatype does NOT violate the relational model's normalization principle. This can be done by only embedding data that belongs to a single parent while keeping data that does not belong to a single parent as referenced data. Referential constraints between JSON datatype based collections can be defined so that users can use JSON duality views to link multiple JSON datatype based document collections as one virtual de-normalized JSON object without any data duplication and update consistency issues. Each JSON document can store an array of identifying references for linking relationships to avoid foreign key joins and maintenance of foreign key indexes. Therefore, there is a need for future work on automatically defining duality views by *analyzing linking and embedding relationships between entities [29], in particular, the implicit referential constraints between JSON datatype based collections*.

11.3 Caching Performance

De-normalization in relational storage is a common technique to boost query performance by materializing join results. Relational materialized views are often used to achieve performance at the cost of materialized view updates whenever base data is changed. JSON datatype can be considered as an advanced de-normalization form as it avoids the entire hierarchical object construction cost that involves many joins. Therefore, JSON is a commonly used datatype in object caches. Unfortunately, it is at the expense of keeping replicated data in the cache synchronized with base data. JSON datatype object caches can be supported in the form of JSON duality materialized views, that are in-memory, on disk, or a combination of both. Therefore, there is a need for future work on automatically defining materialized duality views by *analyzing query and update requirements of data in the application workload*. In particular, the challenge is that a single relational table update can affect many objects from multiple duality views. This fan-out issue requires computation of the closure of objects impacted by the update, which can be non-trivial.

12 Conclusion

There is a historical debate between relational and document models, each of which has its strengths and weaknesses. JSON Relational Duality represents a logical data model abstraction to unify the two physical models inside the database kernel. It is a new approach that architecturally provides the simplicity of use-case-specific solutions with powerful support for implementing multiple use cases. The strength of duality views is that they enable users to independently design application-specific data access and data storage models based on business requirements. The view defines only *what* the mapping between these models should be and that database automatically derives *how* to map these models. Future work includes the design of intelligent tools that can automatically design the data access model (duality view definitions) and data storage model (relational schema with JSON type, if needed) based on workload and business requirements.

Acknowledgments

We thank the Transactions, Data, and JSON teams for their continuous brainstorming sessions, coding deep dives, and astute reviews of this project. We are grateful to the functional and stress testing teams for keeping us honest, and to the performance testing team for their help in evaluating and identifying performance bottlenecks. An impactful project of this stature is, often, a product of continuous quiet efforts by many individuals and teams who put the project before themselves. We are thankful for the privilege of working with so many of them.

References

[1] Oracle 8i Application Developer's Guide Fundamentals Release 2 (8.1.6) Part Number A76939-01. 2000. Instead of Triggers. https://docs.oracle.com/cd/A87860_01/doc/appdev.817/a76939/adg13trg.htm.

[2] Oracle 8 Concepts Release 8.0 A58227-01. 1997. Object Views over Relational Data. https://docs.oracle.com/cd/A58617_01/server.804/a58227/ch_objvw.htm.

[3] Rakesh Agrawal and Narain H Gehani. 1989. Ode (object database and environment): the language and the data model. *ACM SIGMOD Record* 18, 2 (1989), 36–45.

[4] Thierry Barsalou, Arthur M. Keller, Niki Siambela, and Gio Wiederhold. 1991. Updating Relational Databases through Object-Based Views. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA, May 29-31, 1991*, James Clifford and Roger King (Eds.). ACM Press, 248–257. https://doi.org/10.1145/115790.115831

[5] David Beech. 1988. A foundation for evolution from relational to object databases. In *International Conference on Extending Database Technology*. Springer, 251–270.

[6] Michael Blow, Vinayak Borkar, Michael Carey, Christopher Hillery, Alexander Kotopoulis, Dmitry Lychagin, Radu Preotiuc-Pietro, Panagiotis Reveliotis, Joshua Spiegel, and Till Westmann. 2009. Updates in the aqualogic data services platform. In *2009 IEEE 25th International Conference on Data Engineering (ICDE 2009)*. IEEE, 1431–1442.

[7] Vinayak Borkar, Michael Carey, Dmitry Lychagin, Till Westmann, Daniel Engovatov, and Nicola Onose. 2006. Query processing in the aqualogic data services platform. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*. 1037–1048.

[8] Craig Chasseur, Yinan Li, and Jignesh M. Patel. 2013. Enabling JSON Document Stores in Relational Systems. In *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013*, Angela Bonifati and Cong Yu (Eds.). 1–6. http://webdb2013.lille.inria.fr/Paper%2010.pdf

[9] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. 2004. SQL: 2003 has been published. *ACM SIGMoD Record* 33, 1 (2004), 119–126.

[10] Leonidas Fegaras. 2010. Propagating updates through XML views using lineage tracing. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 309–320.

[11] A.L. Furtado and M.A Casanova. 1985. *Updating Relational Views*. Springer, Berlin, Heidelberg.

[12] Oracle 21 Release Developer Guide. 2021. XMLType Views. https://docs.oracle.com/en/database/oracle/oracle-database/21/adxdb/XMLType-views.html.

[13] Hibernate. 2001. Object/Relational Mapping. https://hibernate.org/orm/.

[14] JPAB 2010. JPA Performance Benchmark. https://www.jpab.org/.

[15] Arthur M. Keller. 1985. *Updating relational databases through views*. Ph. D. Dissertation. Stanford University, USA. https://searchworks.stanford.edu/view/1140803

[16] Arthur M. Keller. 1986. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 467–474. http://www.vldb.org/conf/1986/P467.PDF

[17] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora, and Susan Kotsovolos. 2004. Query Rewrite for XML in Oracle XML DB. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 1122–1133. https://doi.org/10.1016/B978-012088469-8.50098-X

[18] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper5.pdf

[19] Zhen Hua Liu, Beda Christoph Hammerschmidt, Douglas Mcmahon, Hui J. Chang, Ying Lu, Joshua Spiegel, Alfonso Colunga Sosa, Srikrishnan Suresh, Geeta Arora, and Vikas Arora. 2020. Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database. *Proc. VLDB Endow.* 13, 12 (2020), 3059–3071. https://doi.org/10.14778/3415478.3415534

[20] Zhen Hua Liu, Beda Christoph Hammerschmidt, Douglas Mcmahon, Ying Liu, and Hui Joe Chang. 2016. Closing the functional and Performance Gap between SQL and NoSQL. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 227–238. https://doi.org/10.1145/2882903.2903731

[21] Zhen Hua Liu, Muralidhar Krishnaprasad, James W. Warner, Rohan Angrish, and Vikas Arora. 2007. Effective and efficient update of xml in RDBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 925–936. https://doi.org/10.1145/1247480.1247589

[22] William C. McGee. 1977. The information management system IMS/VS, part I: General structure and operation. *IBM Systems Journal* 16, 2 (1977), 84–95.

[23] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, and Fred Zemke. 2018. The new and improved SQL: 2016 standard. *ACM Sigmod Record* 47, 2 (2018), 51–60.

[24] MongoDB Inc. 2009. MongoDB. https://www.mongodb.com/.

[25] Elizabeth J O'Neil. 2008. Object/Relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 1351–1356.

[26] Oracle. 2017. Oracle Rest Data Services. https://www.oracle.com/database/technologies/appdev/rest.html.

[27] Red Hat. 2001. Hibernate. https://hibernate.org/.

[28] Snowflake. 2018. Semi-structured data types. https://docs.snowflake.com/en/sql-reference/data-types-semistructured.

[29] William Spoth, Ting Xie, Oliver Kennedy, Ying Yang, Beda Christoph Hammerschmidt, Zhen Hua Liu, and Dieter Gawlick. 2018. SchemaDrill: Interactive Semi-Structured Schema Design. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, Carsten Binnig, Juliana Freire, and Eugene Wu (Eds.). ACM, 11:1–11:7. https://doi.org/10.1145/3209900.3209908

[30] The GraphQL Foundation. 2015. GraphQL. https://graphql.org/.

[31] Transaction Processing Performance Council. 1992. Transaction Processing Performance Council Benchmark C. https://www.tpc.org/tpcc/.