Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database

Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Hui Chang, Ying Lu, Josh Spiegel, Alfonso Colunga Sosa, Srikrishnan Suresh, Geeta Arora, Vikas Arora Oracle Corporation

Redwood Shores, California, USA

{zhen.liu, beda.hammerschmidt, doug.mcmahon, hui.x.zhang, ying.lu, josh.spiegel, alfonso.colunga, srikrishnan.s.suresh, geeta.arora, vikas.arora}@oracle.com

ABSTRACT

Both RDBMS and NoSQL database vendors have added varying degrees of support for storing and processing JSON data. Some vendors store JSON directly as text while others add new JSON type systems backed by binary encoding formats. The latter option is increasingly popular as it enables richer type systems and efficient query processing. In this paper, we present our new native JSON datatype and how it is fully integrated with the Oracle Database ecosystem to transform Oracle Database into a mature platform for serving both SQL and NoSQL style access paradigms. We show how our uniquely designed Oracle Binary JSON format (OSON) is able to speed up both OLAP and OLTP workloads over JSON documents.

PVLDB Reference Format:

Z. Hua Liu et al.. Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database. PVLDB, 13(12) : 3059-3071, 2020.

DOI: https://doi.org/10.14778/3415478.3415534

1. INTRODUCTION

JSON has a number of benefits that have contributed to its growth in popularity among database vendors. It offers a schema-flexible data model where consuming applications can evolve to store new attributes without having to modify an underlying schema. Complex objects with nested master-detail relationships can be stored within a single document, enabling efficient storage and Further, JSON is human retrieval without requiring joins. readable, fully self-contained, and easily consumed by popular programming languages such as JavaScript, Python, and Java. As a result, JSON is popular for a broad variety of use cases including data exchange, online transaction processing, online data analytics.

OLTP for JSON: NoSQL vendors, such as MongoDB [11] and Couchbase [4] provide JSON document storage coupled with simple NoSQL style APIs to enable a lightweight, agile development model that contrasts the classic schema-rigid SQL approach over relational data. These operational stores provide create, read, update and delete (CRUD) operations over

Proceedings of the VLDB Endowment, Vol. 13, No. 12 ISSN 2150-8097.

DOI: https://doi.org/10.14778/3415478.3415534

collections of schema-flexible document entities. This contrasts traditional relational databases which support similar operations but over structured rows in a table. However, over the past decade, many relational database vendors such as Oracle [29], Microsoft SQL Server [10], MySQL [12], PostgreSQL [16] have added support for storing JSON documents to enable schemaflexible operational storage.

OLAP for JSON: Both SQL and NoSQL databases have added support for real-time analytics over collections of JSON documents [4, 16, 15]. In general, analytics require expressive and performant query capabilities including full-text search and schema inference. SQL vendors, such as Oracle [28] are able to automatically derive structured views from JSON collections to leverage existing SQL analytics over JSON. The SQL/JSON 2016 standard [21] provides comprehensive SQL/JSON path language for sophisticated queries over JSON documents. NoSQL users leverage Elastic Search API [8] for full text search over JSON documents as a basis of analytics. All of which have created online analytical processing over JSON similar to the classical OLAP over relational data.

While well suited for data exchange, JSON text is not an ideal storage format for query processing. Using JSON text storage in a database requires expensive text processing each time a document is read by a query or is updated by a DML statement. Binary encodings of JSON such as BSON [2] are increasingly popular among database vendors. Both MySQL [12] and PostgreSQL [16] have their own binary JSON formats and have cited the benefits of binary JSON for query processing. Oracle's in-memory JSON feature that loads and scans Oracle binary JSON (OSON) inmemory has shown better query performance compared with JSON text [28]. In addition to better query performance, binary formats allow the primitive type system to be extended beyond the set supported by JSON text (strings, numbers, and booleans).

Supporting a binary JSON format only to enable efficient query processing and richer types is not enough for OLTP use cases. In such cases, it is critical that applications can also efficiently create, read, and update documents as well. Efficient updates over JSON are especially challenging and most vendors resort to replacing the entire document for each update, even when only a small portion of the document has actually changed. Compare this to update operations over relational data where each column can be modified independently. Ideally, updates to JSON documents should be equally granular and support partial updates in a piecewise manner. Updating a single attribute in a large JSON document should not require rewriting the entire document.

In this paper, we describe the native JSON datatype in Oracle Database and how it is designed to support the efficient query,

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

update, ingestion, and retrieval of documents for both OLTP and OLAP workloads over JSON. We show how fine-grained updates are expressed using the new JSON_TRANSFORM() operator and how the underlying **OSON binary format is capable of supporting these updates without full document replacement.** This results in update performance improvements for medium to large JSON documents.

We will show how data ingestion and retrieval rates are improved by keeping OSON as the network exchange format and adding native OSON support to existing client drivers. These drivers leverage the inherent read-friendly nature of the format to provide "in-place", efficient, random access to the document without requiring conversions to intermediate formats on the server or client. OSON values are read by client drivers using convenient object-model interfaces without having to first materialize the values to in-memory data structures such as hash tables and arrays. This, coupled with the natural compression of the format, results in a significant improvement in throughput and latency for simple reads. We will show how ingestion rates are not hindered by the added cost of client document encoding but instead tend to benefit from reduced I/O costs due to compression.

In this paper, we also present the set of design principles and techniques used to support JSON datatype in the Oracle Database eco-system. The design is driven by variety of customer use cases, including pure JSON **document storage usecases** to process both OLTP (put/get/query/modify) and OLAP (ad-hoc query report, full text search) operations, **hybrid usecases** where JSON is stored along-side relational to support flexible fields within a classic relational schema, JSON generation usecases from relational data via SQL/JSON functions, and JSON **shredding usecases** where JSON is shredded into relational tables or materialized views. Both horizontal scaling via Oracle sharding and vertical scaling via Oracle ExaData and In-Memory store have been leveraged to support all these cases efficiently. The main contributions of this paper are:

- 1. The OSON binary format to support the efficient query, update, ingestion, and retrieval of JSON documents. To the best of our knowledge, OSON is the first binary JSON format that supports general piecewise updates and efficient in-place server and client-side navigation without sacrificing schema-flexibility. The novel design enables queries and updates to be done in logarithmic rather than linear running time.
- 2. The JSON_TRANSFORM() operator provides declarative partial updates over JSON documents in a way that is amenable to efficient piece-wise evaluation over OSON.
- 3. Integration of the JSON datatype with all the salient features of Oracle Database to achieve high performance for both OLTP and OLAP workloads. In particular, the in-memory path-value index format and inverted keyword hash index format for JSON_EXISTS() and JSON_TEXTCONTAINS() in memory predicate evaluation for OLAP is novel.
- 4. An extensive performance study of the benefits of using OSON storage over JSON text for both server and client.

The rest of the paper is organized as follows. Section 2 gives an overview of JSON datatype functionality. Section 3 describes its design. Section 4 is on support of JSON OLTP and OLAP workloads. Section 5 is on performance experiments. Section 6 is

on related work. Section 7 is on future work. Section 8 is conclusion with acknowledgments in section 9.

2. JSON DATATYPE FUNCTIONALITY2.1 SQL/JSON 2016

The SQL/JSON 2016 [21] standard defines a set of SQL/JSON operators and table functions to query JSON text and generate JSON text using VARCHAR2/CLOB/BLOB as the underlying storage. JSON VALUE() selects a scalar JSON value using a path expression and produces it as a SQL scalar. JSON QUERY() selects a nested JSON object or array using a path expression and returns it as a JSON text. JSON TABLE() is a table function used in the SQL FROM clause to project a set of rows out of a JSON object based on multiple path expressions that identify rows and columns. JSON EXISTS() is used in boolean contexts, such as the SQL WHERE clause, to test if a JSON document matches certain criteria expressed using a path expression. These JSON query operators accept SQL/JSON path expressions that are used to select values from within a document. The SQL/JSON path language is similar to XPath and uses path steps to navigate the document tree of objects, arrays, and scalar values. Each step in a path may optionally include predicates over the values being selected. Like XPath, SQL/JSON path leverages a sequence data model and the intermediate result of any SQL/JSON path expression is a sequence of JSON values (objects, arrays, scalars). While the mechanics of SQL/JSON path follows XPath, the syntax is more similar to JavaScript.

2.2 JSON Datatype

In Oracle Database 20c, the "JSON" type can be used to store JSON data instead of VARCHAR/CLOB/BLOB. The JSON type data model is closely aligned with JSON text and includes objects, arrays, strings, numbers, true, false, and null. But like other JSON formats [2], the data model is also extended with SQL primitive types for packed decimal, IEEE float/double, dates, timestamps, time intervals, and raw values. We refer to this logical data model as the JSON Document Object Model (JDOM). The OSON binary format for JSON datatype is a serialization of a JDOM. SQL/JSON 2016 supports type casting item functions, such as .number(), .string(), .date(), .binary() etc, that can cast string to non-string built-in datatypes.

The JSON datatype can be used as the type of a table column, view column, parameter, return value, or a local variable datatype in SQL and PL/SQL functions. The SQL/JSON operator JSON_QUERY() by default returns JSON datatype. JSON_TABLE() can return JSON datatype as projected column datatype. SQL/JSON generation functions can return JSON datatype. All of these have overcome the limitation from the JSON text with IS JSON constraint based pseudotype which in some cases may lose type information between SQL operations. Implicit conversion between JSON datatype and JSON text is supported by the SQL compiler.

Figure 1 shows a set of SQL/JSON queries and DML statements over a *purchaseOrder* table whose DDL definition is shown as D1. The JSON type column *jdoc* stores a purchase order JSON document. DML statement 11 shows an example of a JSON document representing a purchase order being inserted into the *purchaseOrder* table. In 11, the compiler will implicitly wrap the JSON string within a JSON() constructor that encodes JSON text to OSON binary during insertion.

```
Q4 SELECT jdoc
D1 CREATE TABLE purchaseOrder
      (did NUMBER PRIMARY KEY, jdoc JSON)
                                                                    FROM purchaseOrder
I1 INSERT INTO purchaseOrder
    VALUES (1, ' {"purchaseOrder": {
    "podate": "2015-06-03",
     "shippingAddress": {"street": "3467 35th Ave",
"city" : "Berkeley", "state": "CA", "zip":
    94612},
     "comments" : "Discounted sales Independence Day",
     "sparse_id" :"CDEG35",
     "items": [
        {"name" : "TV", "price": 345.55, "quantity": 2,
          "parts": [
            {"partName": "remoteCon", "partQuantity": 1},
            {"partName": "antenna", "partQuantity": 2}]},
         {"name": "PC", "price": 446.78, "quantity": 10,
           "parts": [
            {"partName": "mouse", "partQuantity": 2},
            {"partName": "keyboard", "partQuantity": 1}]}
                                                                    }
     ]}}');
Q1 SELECT did.
      po.jdoc.purchaseOrder.podate.date(),
      po.jdoc.purchaseOrder.shippingAddress,
      po.jdoc.purchaseOrder.items[*].count(),
      po.jdoc.purchaseOrder.item[1]
    FROM purchaseOrder po
    WHERE po.jdoc.purchaseOrder.podate.date() =
    TO_DATE('2015-06-03','YYYY-MM-DD') AND
po.jdoc.purchaseOrder.shippingAddress.zip.number()
BETWEEN 94610 AND 94620
                                                                       }
Q2 SELECT did.
      JSON_QUERY(jdoc,
          $.purchaseOrder.items?(@.price > 300)'),
       JSON_VALUE(jdoc,
          $.purchaseOrder?(exists(@..parts?(@.partName ==
          "mouse" && @.partQuantity >=2 ))).podate')
    FROM purchaseOrder po
    WHERE
      JSON_EXISTS(jdoc,
         '$.purchaseOrder.sparse_id?(@ == "CDEG35")') AND
      JSON EXISTS(idoc.
         '$.purchaseOrder.items?(@.name == "TV" &&
          @.parts.partQuantity >= 2)')
Q3 SELECT po.did, jt.*
    FROM purchaseOrder po, JSON_TABLE (jdoc
    COLUMNS (
     poid NUMBER PATH '$.purchaseOrder.id',
     podate DATE PATH '$.purchaseOrder.podate',
                PATH '$.purchaseOrder.sparse_id',
     sp id
     NESTED '$.purchaseOrder.items[*]'
      COLUMNS (name, price NUMBER, quantity NUMBER),
                                                                     "mouse"
      NESTED '$.parts[*]
        COLUMNS (partName, partQuantity NUMBER ))) jt
```

```
WHERE JSON_EXISTS(jdoc,
       `$.purchaseOrder.items?(
         @.price > $price && @.quantity >= $qty &&
         (exists(@.parts?(
            @.partName == $pname &&
            @.partQuantity >= $pquantity))))'
      PASSING TO_NUMBER(:1) AS "price",
            TO_NUMBER(:2) AS "qty",
             :3 AS "pname"
            TO_NUMBER(:4) AS "pquantity")
    AND JSON TEXTCONTAINS(jdoc,
        `$.purchaseOrder.comments',
        '{Independence} NEAR {discount}')
Q5 select json {
       'name' : li.itemName,
'sales' : li.price * li.quantity
    FROM lineItems_rel li
Q6 SELECT
      JSON {
         'id'
                  : po.Id,
        'poDate' : po.podate,
        'items' : (SELECT JSON_ARRAYAGG(JSON {*})
                     FROM lineItems_rel E
```

```
WHERE E.fid_po = po.Id)
```

```
FROM PurchaseOrder_rel po
```

```
U1 \ \mbox{UPDATE} purchaseOrder po
    SET jdoc = JSON_TRANSFORM(jdoc,
      REPLACE
     $.purchaseOrder.shippingAddress.city'
                = 'Oakland',
       REPLACE '$.purchaseOrder.shippingAddress.zip'
                = 94607,
       SET '$.purchaseOrder.contactPhone' =
        JSON('["(415)-667-8960","(510)332-8888"]'),
       REMOVE '$.purchaseOrder.sparse_id',
      APPEND '$.purchaseOrder.items' =
              JSON('{"items" :[{"name":"iphone",
"price" : 635.54, "quantity" :2}]}'))
    WHERE po.jdoc,purchaseOrder.podate.date() =
    T0_DATE('2015-06-03')
Q7 SELECT did,
      JSON_QUERY(jdoc,
          $.purchaseOrder.items?(@.price > 300)'),
       JSON_VALUE(jdoc,
          '$.purchaseOrder?(@..parts?(@.partName ==
```

```
&& @.partQuantity >=2 )).podate')
FROM purchaseOrder po
WHERE EXISTS(
   SELECT 1
   FROM MV PO
   WHERE MV_PO.FID = po.did AND
          MV_PO.sp_id = 'CDEG35' AND
MV_PO.name = 'TV' AND
          MV_PO.quantity >= 2)
```

Figure 1. Example SQL/JSON queries and DMLS statements

```
'JSON_TRANSFORM' '('
       input_expr ',' operation (',' operation)*
JSON_TRANSFORM_returning_clause?
       JSON_passing_clause?
operation := (removeOp | insertOp| replaceOp |
   appendOp | setOp | renameOp | keepOp)
removeOp := 'REMOVE' pathExp
    (('IGNORE' | 'ERROR') 'ON' 'MISSING')?
insertOp := 'INSERT' pathExp '=' rhsExpr
 (('REPLACE' | 'IGNORE' | 'ERROR') 'ON' 'EXISTING')?
 (('NULL' | 'IGNORE' | 'ERROR' | 'REMOVE') 'ON'
 'NULL')?
replaceOp := 'REPLACE' pathExp '=' rhsExpr
(('CREATE' | 'IGNORE' | 'ERROR') 'ON' 'MIS
(('NULL' | 'IGNORE' | 'ERROR' | 'REMOVE')
                                                                     'MISSING')?
        'NULL')?
appendOp := 'APPEND' pathExp '=' rhsExpr
(('CREATE' | 'IGNORE' | 'ERROR') 'ON' 'MISSING')?
(('NULL' | 'IGNORE' | 'ERROR') 'ON' 'NULL')?
setOp := 'SET' pathExp '=' rhsExpr
 (('IGNORE' | 'ERROR' | 'REPLACE') 'ON' 'EXISTING')?
 (('CREATE' |'IGNORE' | 'ERROR') 'ON' 'MISSING')?
 (('NULL' | 'IGNORE' | 'ERROR') 'ON' 'NULL')?
renameOp := 'RENAME' pathExpr 'WITH' stringLiteral
   (('IGNORE' | 'ERROR') 'ON' 'MISSING')?
keepOp := 'KEEP' (pa
'ON' 'MISSING')? )
              = 'KEEP' (pathExpr (('IGNORE' | 'ERROR')
   (',' pathExpr (('IGNORE' | 'ERROR')
'ON' 'MISSING')? )*
rhsExpr := sqlExpr ('FORMAT JSON')?
```

Figure 2. JSON_TRANFORM grammar

2.3 Simplified Syntax for SQL/JSON

Oracle Database provides a simplified syntax for querying JSON values as an alternative to calling more expressive but verbose SQL operators such as JSON_QUERY() and JSON_VALUE(). A simple JSON path navigation without any predicates can be abbreviated using the dot notation as shown in the select list of Q1. Tailing step functions such as number(), binary(), date(), and timestamp() can be used to specify the return type of the expression. For example, po.jdoc.purchaseOrder.podate. date(), is internally translated into JSON_VALUE(po.jdoc, '\$.purchaseOrder.podate' RETURNING DATE). The trailing step function casts the value to the corresponding SQL built-in type (in this case, DATE). Without a trailing step function, the return type of the is JSON. For expression example. po.jdoc.purchaseOrder.shipping, is translated into JSON_QUERY(po.idoc, '\$.purchaseOrder.shipping' RETURNING JSON).

Other types of trailing step functions are also supported. For example, po.jdoc.purchaseOrder.items[*].count() in Q1 illustrates a sequence item aggregation function that returns the total number of items in an array. This avoids using the general purpose JSON_TABLE() operator for simple aggregations.

Examples Q5 and Q6 show the simplified syntax for J50N_OBJECT(). This JSON constructor syntax allows users to create new JSON objects and arrays using a syntax that resembles

the JSON that the expression creates. In Q6, the expression JSON{*} automatically creates a JSON object representation of the relational row. There are other syntax simplifications for accessing JSON as well, all of which map to the core SQL/JSON operators [20,21].

2.4 Updating with JSON TRANSFORM

JSON TRANSFORM() is a new SQL operator to declaratively apply a sequence of modification operations to JSON values selected by the SQL/JSON path language. Figure 2 shows the grammar for the JSON_TRANSFORM() operator. The INSERT operation inserts a new value into a JSON object or array. REPLACE operation replaces an existing value with a new value. The APPEND operation appends a new value into an array. The SET operation either replaces an existing value by a new value or adds a new value if the original one does not exist. REMOVE operation removes values. U1 in Figure 1 is an example of a SQL UPDATE statement that uses JSON TRANSFORM to modify a JSON column value. Logically, the update is a full replacement of the existing value with the However internally, the update is applied modified value. piecewise without materializing the new and old value at once (see section 3.4). JSON_TRANSFORM can also be used in the select list of a query to perform a transformation on a JSON value without changing the persistent storage. For example, JSON_TRANSFORM could be used to redact a social security number using a REMOVE operation before sending the document to the client. The KEEP operation is the inverse of REMOVE: only values identified by path expressions are kept and all others are removed.

2.5 Client JSON Datatype Access

Both JDBC (Java Database Connectivity) and OCI (Oracle Call Interface) have been enhanced with OSON support. Using these libraries, applications can read, create, and modify JSON type values, convert values between JSON text and OSON, and store and retrieve values in the database. These libraries provide simple object model access APIs to randomly navigate and access values within a document. For example, the package oracle.sql.json in JDBC implements JSR374/JSON-P interfaces (javax.json.*) over OSON and is compatible with JSR367/JSON-B so that user domain objects can be mapped directly to and from JSON type values without incurring JSON text parsing or serialization costs.

```
PreparedStatement select = con.prepareStatement
    ("SELECT jcol FROM purchaseOrder WHERE did =1");
ResultSet rs = select.executeQuery();
rs.next();
JsonObject doc =
    rs.getObject(1, javax.json.JsonObject.class);
System.out.println(doc.getString("sparse_id"));
rs.close();
```

Figure 3. JSON datatype in JDBC

Figure 3 is an excerpt from of a Java program that uses JDBC to retrieve a purchase order document from the server. In this case, the OSON bytes for the purchase order are transferred directly to the application and exposed to the developer using the standard javax.json.JsonObject interface. The value for sparse_id is read from the object in-place without processing or converting the rest of the document. This is discussed more in sections 3 and 4.

<pre>oson_document := header dictionary tree_seg</pre>								
<pre>(extended_tree_seg) dictionary := sortedFieldHashes fieldNameOffsets fieldNames</pre>								
<pre>tree_seg := value+</pre>								
extended_tree_seg := tree-seg								
value := object array scalar forward_offset								
<pre>object := OBJECT_CODE (fields delegate_offset)</pre>								
offsetArray								
fields := size sortedFidArray								
array := ARR_CODE size offsetArray								
scalar := scalarCode sqlScalar								

Figure 4. OSON Binary Format (pseudo grammar)

2.6 Simple Oracle Document Access (SODA)

SODA [19] is Oracle's simple document access API for developers to use Oracle Database as a pure JSON document store. This API provides NoSQL-style access (CRUD) based on the document's ID, presenting a key/value model similar to other common document stores. With SODA, data is logically managed by an application using collections of simple documents rather than tables containing rows. However, SODA collections are still backed by internally managed relational tables having ID, JSON, and other metadata columns. With this model, applications can be written without using SQL but SQL/JSON can still be leveraged over collection data for analytics and reporting if ever More complex queries over JSON documents are needed. specified as QBEs (Query By Example). QBE's are JSON documents themselves which are internally transformed to equivalent SQL/JSON queries over the underlying JSON datatype column of the table. SODA is available in all popular client programming languages: Java, Node.js, Python, C, PL/SQL.

3. JSON DATATYPE DESIGN

3.1 JSON Datatype Derivation from BLOB

The JSON datatype is internally designed as a derivation of the BLOB datatype for storing OSON bytes. This simple approach enables Oracle to provide complete implementation support for the JSON datatype in every part of Oracle's ecosystems within a yearly based release. However, this is all transparent to developers because the general LOB APIs, to read and write LOBs using offsets, cannot be used on JSON type. One main advantage of being a SQL datatype is that SQL static typing can enforce desired type semantics over JSON. And in contrast, with JSON text storage developers use the more complicated LOB API explicitly to access and modify JSON.

Classic BLOB was initially designed to store large binary objects and provide random access over any file or media (e.g. a music file). However, JSON documents for operational data are typically smaller (kilobytes to megabytes). To achieve optimal performance, we inline OSON values up to the database block size to avoid out of row BLOB access as much as possible. This allows SQL/JSON evaluation to directly access OSON bytes as if it were normal relational RAW column inside a row. When an OSON value is larger than the block size, it is stored outside of the row using multiple data blocks managed by the Oracle securefile i-node infrastructure [13]. Data blocks for OSON BLOB storage are lazily read and cached in the buffer cache based on tree navigation patterns instead of linearly reading everything. For large OSON, we only need to read OSON data blocks that are relevant to answer the path query.

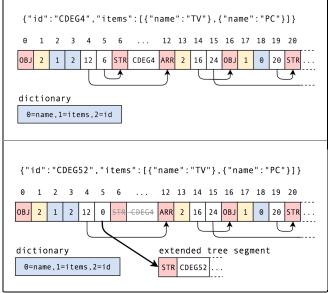


Figure 5. OSON example

For a BLOB storing OSON bytes for the JSON datatype, we applied value based LOB semantics on both the server and client. A large client side prefetch buffer is auto-configured for OSON BLOB. The lifecycle for OSON BLOB is restricted to cursor fetch duration in order to prevent resource leakage.

3.2 OSON Binary Format Design

The following section describes the OSON binary format and its characteristics. Space precludes giving a full formal definition of OSON but Figure 4 gives a pseudo-grammar that identifies the salient aspects of its structure. This section gives an overview of the structure and highlights its benefits.

3.2.1 OSON Structure

An OSON image consists of a header, a dictionary, and a value segment. The header gives a fixed signature specific to OSON and records various aspects about the image that are needed by a reader, such as the size of byte offsets used within the document.

The dictionary contains the set of distinct field names used within all objects in the document. Specifically, it first stores a sorted array of hash codes for each distinct key (sortedFieldHashes). This array is followed by a second array of equal length (fieldNameOffsets) that stores the corresponding offset of the key string. These offsets reference a third array (fieldNames) that contain the actual field strings. The unique *field ID* of a given key is its ordinal position within fieldNameOffsets.

The tree value segment follows the dictionary and encodes the objects, arrays, and primitive values used within the document. Objects are encoded by a set of size-prefixed parallel arrays that encode the entries in the object. The first array (sortedFidArray) gives the sorted field IDs of the field names in the object. The second array (offsetArray) gives the corresponding offsets to values in the value segment. Arrays are encoded by a single size-prefixed offset array containing the offsets of the values in the array. The upper part of Figure 5 shows an example of the OSON layout without any partial updates. After a partial OSON update, extended tree segment is appended as show in the bottom half of Figure 5. The update replaces the string 'CDEG4' with 'CDEG52'. Note if 'CDEG4' is replaced by 'CDEG8', then a

direct replacement is done without appending in the extended tree segment.

3.2.2 Self-contained and platform independent

Similar to JSON text and BSON, OSON does not depend on any external schema information and supports unencumbered schema evolution. This is a critical property to support distributed computing over the data without synchronizing on central schema access. Many database features, like partitioning, replication, sharding, import/export, and transportable tablespace require data to be self-contained and accessible on any platform without data conversion.

3.2.3 Compact

Unlike JSON text and BSON, OSON maintains a local dictionary in the header that encodes each distinct field name. For JSON documents having many repeating field names due to arrays of identical JSON object structures or recursive JSON object structures, OSON is often much smaller than the equivalent UTF-8 encoded JSON text or BSON since the repeated field names are replaced by small field IDs. Furthermore, multiple objects containing the same field names will only have their field ID array stored once. Objects can reference the offset of the other object that shares the same structure (delagate_object) instead of repeating the same field ID array. For example, in the purchaseOrder JSON document shown in Figure 1, there is an 'item' array, each of which has an identical object structure of 'name', 'price', 'quantity', 'parts' fields. Each 'part' array has identical object structures of 'partName', 'partQuantity' fields. These repeating field IDs are stored just once and reused by multiple objects. The field ID size is fixed within an OSON document as either 1 or 2 or 4 bytes depending on how many distinct field names there are in the document. Commonly, documents have less than 256 distinct keys and a 1-byte field ID is thus sufficient. Similarly, the size of value offsets used to reference values will be either 2 or 4 bytes depending on whether the total size of the OSON exceeds 64K. Both the field ID and offset sizes in effect are encoded in the OSON header.

3.2.4 Efficient in-place navigation

Tree navigation is performed in-place over the OSON bytes without first loading the data into other in-memory structures. A value offset is used as direct pointer to the OSON bytes that encode that value. Searching for a key name within an object, or an array indexed element uses *jump navigation* to efficiently skip past irrelevant portions of the document. Specifically, when searching for a given key within an object, a binary search over sortedFidArray is performed to identify the offset in the corresponding offsetArray. This offset can be used to *jump* directly to the corresponding value. Likewise, elements within an array can be efficiently located at any position using the offsetArray. Both the server and client drivers make use of inplace jump navigation to support efficient data access without converting the OSON image to other data structures.

3.2.5 Efficient piece-wise updates

OSON supports partial updates so that changes can be made without replacing the entire document, as would be required for JSON text or BSON. Partial BSON update is limited to the case that the new content length exactly matches the old content length because BSON uses fixed offsets throughout the document which need to be recalculated after a size changing update. Partial update for OSON can handle complex update operations. When the old content length is bigger than or equal to the new content length, it does an in-place update. When in-place update is not feasible, the old value is tombstoned with the forwarding address (forward offset) of the new content that is appended at the end of the original OSON bytes. Subsequent change of the same node but with larger content creates a new forwarding address that is still stored in the original tombstone location to avoid forwarding address chaining. See bottom part of Figure 5. When accumulated appended pieces exceed a certain threshold relative to the original document size, the encoding is re-compacted and the resulting OSON bytes fully replace the old document. Partial update leverages the fact that OSON uses tree offset based jump navigation. To avoid a rebuild of the dictionary due to the insertion of new distinct field names, OSON supports partial dictionary rebuilding by tracking dictionary codes that have been actually changed due to insertion of new distinct field names and then only patching those changed dictionary codes in the end. Compared with the OSON format described in paper [28], this OSON format is enhanced to support piece-wise update of OSON using concept of forward offsets (forward offset) and reduces OSON size for common JSON documents by allowing object encodings to share field structures (delegate_offset).

3.2.6 SQL scalar binary compatibility

All JSON scalar values use the same encoding as existing SQL built-in datatypes. For example, JSON packed decimal is encoded using the same binary format as that of the NUMBER built-in type in Oracle SQL. This allows for efficient conversions and low impedance mismatch when working with JSON inside the database. JSON scalar content is mapped to relational columns and back without loss of fidelity.

3.3 Fast SQL/JSON Path Evaluation

Q1, Q2, and Q4 in Figure 1 show SQL/JSON path language expressions used in JSON_VALUE(), JSON_QUERY(), and JSON_EXIST(). Notice that in addition to simple child steps, Oracle also supports a recursive descendant path step. See "...parts" in Q2 of Table 1 which selects all entries for key "parts" within each descendant object of the current value. Q3 uses JSON_TABLE() to project a master detail hierarchy within a document as a set of flat table rows. The path expressions used within these SQL/JSON operators are evaluated efficiently over the input OSON column value using in-place jump navigation.

When compiling a path expression, a hash function is applied to the field names in the path to generate a corresponding hash code for each field name. These hash codes are stored in the compiled execution plan for the path expression. The OSON field name dictionary (sortedFieldHashes) is organized based on the sorted hash id. At run time, the hash ID from the path is first located within the OSON field name dictionary using a binary search to obtain the corresponding field ID within the document. If the field is not found in the dictionary, it means the field does not occur anywhere in the document. This is particularly helpful for the heterogeneous JSON collection where there are many possible sparse optional fields. Searching for non-existence of fields within JSON text or BSON requires a scan of the entire document to the end whereas OSON only requires a binary search within the OSON field name dictionary. When the object field ID is found in the dictionary, it is then located in the field ID array of the current object, again using binary search, to get the corresponding child offset.

Oracle's built-in SQL datatype format is designed to support type agnostic byte comparable for range comparison. This property is exploited for range comparisons in SQL/JSON path expressions because OSON scalar binary is the same as that of SQL built-in SQL datatype. Also, when the SQL return type used within JSON_VALUE() matches a JSON scalar, its bytes are copied out directly from OSON bytes without any datatype conversion.

3.4 Fast JSON_TRANSFORM() Evaluation

Consider execution of the update statement U1 in Figure 1. Logically, the JSON TRANSFORM() expression on the right-hand side of the SET clause is evaluated over the input JSON value, applying a sequence of modification operations to yield a new temporary document that replaces the JSON column value. However, replacing the entire document in this manner is inefficient as it requires the database to generate a redo log entry proportional to the size of the document. Oracle Database optimizes U1 execution by applying piece-wise updates on the OSON storage without generating a temporary new OSON document to fully replace the previous stored OSON document. With this optimization, transaction redo log size is usually proportional to the size of the change rather than the full document size. At runtime, JSON_TRANSFORM() evaluates the update over the OSON bytes using only 3 low-level update operations: length preserved byte replacement, append bytes at the end and truncate bytes at the end. Typical OSON update operations only result in few database BLOB storage block changes instead of every BLOB storage block being modified. Therefore, update performance using JSON_TRANSFORM() is improved significantly. When there are many partial updates that have accumulated on an OSON document, re-compacting the OSON document to reclaim the space due to deletion occurs automatically. The partial BLOB update API is similar to the POSIX/UNIX file system API [13]. We note this partial OSON update technique is applicable to OSON stored in file systems as well.

4. JSON WORKLOADS

The traditional model of using separate databases for different types of workloads is more and more breaking in favor of socalled *translytic databases* that support both transactional and analytic workloads at once. We have designed JSON type and the SQL/JSON facilities to support both types of workloads over JSON data.

4.1 OLTP Performance

JSON OLTP workloads require high performance document retrieval (i.e. get() operations). A primary key index on a document identifier is required to efficiently retrieve a JSON document by id lookup. For example, the DDL statement D1 in Table 1 shows the primary key specification for column did of the purchaseOrder table. A functional index is needed to facilitate common secondary access paths. Efficient execution of Q1 in Table 1 needs two functional indexes on paths po.jdoc.purchaseOrder.podate.date(), po.jdoc. purchaseOrder.shippingAddress.zip.number(). B+ tree based functional indexes offer the best tradeoff to speed up document retrieval based on range or equality predicates while being both statement and transactionally consistent with respective to underlying DML. DML statements, such as inserts, must compute the functional index expression to maintain the B+ tree for each

operation. Since OSON uses jump navigation to evaluate SQL/JSON path expressions, it provides the best performance to compute the JSON_VALUE() expression during functional index maintenance.

OSON also reduces transfer costs and enables efficient clientaccess to documents. The OSON bytes for the document retrieved are transferred directly to the client application, avoiding any transformation or serialization costs on the server. The OSON bytes are also typically smaller than the corresponding JSON text or BSON value which reduces both IO and network transfer overhead. Like the server, the client application can read the OSON bytes using jump offset navigation, avoiding any transformation costs on the client as well. In contrast, for JSON text or BSON, the application must load the data into alternative data structures such as hash tables to enable efficient random access to the document. Some database vendors use binary JSON as a storage format but then serialize the value to JSON text before sending it to a client. These conversions can consume significant resources on the database server, especially in OLTP type use cases where many clients are using the database at once.

OLTP workloads also require high performance ingestion (i.e. put() operations). Using OSON reduces I/O and network transfer costs during an insert since the OSON value is typically smaller than the corresponding JSON text. Client-side encoding of OSON also reduces server side CPU usage during an insert since the server can directly store the bytes received from the client with OSON bytes verification.

As discussed in section 3.4, document updates are also important to OLTP workloads. OSON has efficient partial OSON update capability that typically result in transaction redo-log size proportional to small changes of OSON instead of the full OSON document replacement. Execution of U1 in **Table 1** uses a functional index to locate the document and then performs partial update.

4.2 OLAP Performance

4.2.1 Incrementally Refreshable Materialized Views Materialized views (MVs) have traditionally been used for materializing aggregations persistently. Depending on the complexity of SQL expressions used in MVs, Oracle allows developers to control whether MVs are refreshed incrementally or fully, and the frequency of refresh at the statement level, at the transaction level or at scheduled intervals. MVs are typically used to speed up OLAP queries. In this paper, we have applied JSON_TABLE() MV to speed up SQL/JSON OLAP queries.

JSON_TABLE() queries, such as the one shown in Q3, can be used to define MV for analytics. The view can be set to refresh at the statement or transaction commit level. This is feasible because the underlying JSON_TABLE() MV table stores the primary key of the JSON datatype column in the original table as a foreign key. In this example, when new JSON document is inserted in the purchaseOrder table, Oracle MV maintenance runs JSON_TABLE() over the new JSON document to return a set of relational rows tagged with the primary key of the new document and then inserts them into the MV table. JSON_TABLE() evaluation over OSON is more efficient than JSON text for the reasons discussed in section 3. When JSON documents are deleted from the purchaseOrder table, Oracle MV maintenance deletes those rows in MV table whose foreign key value matching the primary keys of the deleted rows. Update of JSON datatype column in purchaseOrder table is triggered as deletion followed by insertion operations in the MV maintenance layer. The JSON_TABLE() MV maintenance can be triggered at individual DML statement completion time. This statement level consistency is semantically equivalent to that of index maintenance during which a session can see its own change immediately without committing the transaction. The MV table update is rolled back if the transaction containing these DML statements is rolled back.

Due to the MV statement level consistency semantics, JSON_TABLE() based MVs can be used to transparently rewrite not only queries that use JSON TABLE() in the FROM clause but also queries that use JSON EXISTS() as predicate in WHERE clause. For example, assume that MV PO is a MV defined over the JSON_TABLE() query given in Q3. The query given in Q2 can be internally rewritten to the query given in Q7. Q7 uses MV_P0 to help evaluate the query over the purchaseOrder table. The JSON_EXIST() expressions in the original Q2 are transformed in Q7 into a semi-join using an EXISTS subquery. MV_P0.FID is the internal foreign key of the MV table that joins to the primary key did in the purchaseOrder table. Semi-joins like this one are amenable query optimizations for efficient execution plans. For example, when MV_P0 has secondary indexes on sp_id, name and quantity columns or if the table is placed in the Oracle in-memory columnar store, significant performance improvement is achievable.

4.2.2 Parallel Query

SQL/JSON OLAP queries can be executed in parallel with multiple slaves executing SQL/JSON operators used in the select, where, group by and order by clauses. JSON_TABLE() is executed in the same slave process which reads the JSON column. SQL/JSON operators in the select list are pushed down to each slave process so that they are executed in parallel. Slave processes add intermediate results as temporary OSON BLOBs in the parallel table queue. The query coordinator processes the queue and ships the final results to DB client.

4.2.3 In-memory Indexes

Although JSON search index is powerful, it is based on disk structures that inevitably impact DML performance. Oracle DB in-memory store[22], however, is able to maintain an alternative format of data for fast query without paying the disk I/O cost to persist them while still providing transactional consistency.

The Oracle In-Memory store has an in-memory expression feature (IME) [1] that enables storing virtual column expressions in memory. JSON_VALUE() is used to shred the JSON into multiple columns and load them into an in memory columnar format that is amendable to SIMD scans [28]. JSON datatype is further integrated with IME as a memory pool to maintain a compact JSON search index based memory representation so that JSON EXISTS() and JSON TEXTCONTAINS() predicates can be evaluated in-memory. This is achieved by using both a pathvalue index format to support JSON_EXISTS() and a inverted keyword hash index format to support JSON_TEXTCONTAINS(). The Oracle in-memory store assigns each row an ordinal number within a memory unit, and we use that ordinal number as the document id (docid) for keyword inverted-list and path-value indexing. The hash index maps keywords into posting lists having that keyword using a serialized in-memory hash table. The posting list is a bitmap for docid with the doc offset. Path-Value inmemory maps a 16 byte hash id for each unique path to a sorted distinct leaf scalar values, each of which maps to a bitmap for the docid having that scalar value.

If the in-memory index fails to load in an IME unit due to lack of memory or if it cannot keep up with heavy DML rates, execution automatically falls back to use OSON bytes for evaluation. Unlike disk-based indexes, the in-memory index maintenance does not slow down the main DML execution path because its population is trigged periodically and is done in the background.

4.2.4 Predicate Scans in Exadata Storage Cells

Oracle Exadata [14] is an engineered system for Oracle Database. A key aspect of the Exadata architecture is to separate compute and storage and push predicate evaluation down to storage server. SQL predicates evaluated during a table scan are pushed down and evaluated in the storage layer by parallel kernel threads. The storage layer returns matching rows. This improves the performance of scans by orders of magnitudes by reducing the amount of data sent back to compute nodes. JSON support takes full advantage of this by supporting SQL/JSON functional evaluations in the storage cells, so that SQL/JSON predicates, such as JSON_EXISTS() and JSON_VALUE() in WHERE clauses are pushed down to storage cell server for evaluation when any indexing mechanism in the compute node are not available or determined to be non-beneficial by optimizer.

4.3 Sharding & Full Text Search

This section covers sharding and full-text search which are common features of NoSQL databases and useful for improving both operational and analytical workloads.

4.3.1 Full-text Search

Oracle Database supports the operator JSON_TEXTCONTAINS(), usable in WHERE clause to facilitate JSON path context aware full text search. For example, Q4 in Figure 1 uses JSON_TEXTCONTAINS() to perform the full text search '{Independence} NEAR {discount}' under a SQL/JSON path \$.purchaseOrder.comments. OLAP queries over JSON data typically have ad-hoc path value range searches that are not possible to create many functional indexes or statement refreshable MVs to capture all possible paths in a JSON datatype column. Therefore, Oracle JSON Search Index is designed to index everything in a JSON datatype column, both full text and leaf scalar values. The SQL compiler detects all SQL/JSON operators: JSON_TEXTCONTAINS(), JSON_EXISTS(), JSON_VALUE() in WHERE clause over the same JSON datatype column and then combines them into one tree pattern search operator to be evaluated by the JSON search index.

There are two components in the JSON search index. One component indexes keyword tokens of both scalar strings and hierarchical tree structures. It uses an inverted index layout to map each keyword token to its document ids and its containing hierarchical path as intervals. The other component indexes leaf number and timestamp values and its leading JSON path. It has B+ tree index to index a combination of 16 bytes hash of each unique JSON paths and its distinct leaf scalar node value for path-value range search. The posting list is compact because the posting list uses ordinal numbers as document ids and does delta compression of them. Furthermore, ordinal number based document ids also facilitate fast bitmap joins and pre-sorted merge

Table 1. Document sizes

UTF8 (b)	BSON (b)	OSON (b)	Dict	vsUTF8	vsBSON	#Object	#Keys	#Array	#String	#Number
613	764	524	5%	0.9x	0.7x	20	33/5	1	31	0
1,782	1,813	1,950	30%	1.1x	1.1x	4	56/55	0	31	21
2,608	3,094	2,160	16%	0.8x	0.7x	26	100/32	14	124	1
2,943	3,293	2,476	6%	0.8x	0.8x	46	100/19	14	87	0
8,842	8,440	5,591	19%	0.6x	0.7x	38	307/74	29	107	56
40,285	37,526	20,486	18%	0.5x	0.5x	81	1,435/246	9	572	476
76,861	75,195	38,383	11%	0.5x	0.5x	490	3,300/282	23	1,263	1,129
141,051	133,307	103,897	0%	0.7x	0.8x	1,688	6,620/40	52	652	6,860
682,228	No Data	483,053	0%	0.7x	No Data	9,727	36,778/42	42	666	40,375
3,374,379	3,303,387	2,167,101	0%	0.6x	0.7x	14,712	112,356/90	12,738	43,927	27,618
41,548,995	37,352,414	13,801,333	0%	0.3x	0.4x	100,141	1,839,847/73	1	100,143	901,263
	613 1,782 2,608 2,943 8,842 40,285 76,861 141,051 682,228 3,374,379	613 764 1,782 1,813 2,608 3,094 2,943 3,293 8,842 8,440 40,285 37,526 76,861 75,195 141,051 133,307 682,228 No Data 3,374,379 3,303,387	613 764 524 1,782 1,813 1,950 2,608 3,094 2,160 2,943 3,293 2,476 8,842 8,440 5,591 40,285 37,526 20,486 76,861 75,195 38,383 141,051 133,307 103,897 682,228 No Data 483,053 3,374,379 3,303,387 2,167,101	613 764 524 5% 1,782 1,813 1,950 30% 2,608 3,094 2,160 16% 2,943 3,293 2,476 6% 8,842 8,440 5,591 19% 40,285 37,526 20,486 18% 76,861 75,195 38,383 11% 141,051 133,307 103,897 0% 682,228 No Data 483,053 0% 3,374,379 3,303,387 2,167,101 0%	613 764 524 5% 0.9x 1,782 1,813 1,950 30% 1.1x 2,608 3,094 2,160 16% 0.8x 2,943 3,293 2,476 6% 0.8x 8,842 8,440 5,591 19% 0.6x 40,285 37,526 20,486 18% 0.5x 76,861 75,195 38,383 11% 0.5x 141,051 133,307 103,897 0% 0.7x 682,228 No Data 483,053 0% 0.7x 3,374,379 3,303,387 2,167,101 0% 0.6x	613 764 524 5% 0.9x 0.7x 1,782 1,813 1,950 30% 1.1x 1.1x 2,608 3,094 2,160 16% 0.8x 0.7x 2,943 3,293 2,476 6% 0.8x 0.8x 8,842 8,440 5,591 19% 0.6x 0.7x 40,285 37,526 20,486 18% 0.5x 0.5x 76,861 75,195 38,383 11% 0.5x 0.5x 141,051 133,307 103,897 0% 0.7x 0.8x 682,228 No Data 483,053 0% 0.7x No Data 3,374,379 3,303,387 2,167,101 0% 0.6x 0.7x	613 764 524 5% 0.9x 0.7x 20 1,782 1,813 1,950 30% 1.1x 1.1x 4 2,608 3,094 2,160 16% 0.8x 0.7x 26 2,943 3,293 2,476 6% 0.8x 0.8x 46 8,842 8,440 5,591 19% 0.6x 0.7x 38 40,285 37,526 20,486 18% 0.5x 0.5x 81 76,861 75,195 38,383 11% 0.5x 0.5x 490 141,051 133,307 103,897 0% 0.7x 0.8x 1,688 682,228 No Data 483,053 0% 0.7x No Data 9,727 3,374,379 3,303,387 2,167,101 0% 0.6x 0.7x 14,712	613 764 524 5% 0.9x 0.7x 20 33/5 1,782 1,813 1,950 30% 1.1x 1.1x 4 56/55 2,608 3,094 2,160 16% 0.8x 0.7x 26 100/32 2,943 3,293 2,476 6% 0.8x 0.8x 46 100/19 8,842 8,440 5,591 19% 0.6x 0.7x 38 307/74 40,285 37,526 20,486 18% 0.5x 0.5x 81 1,435/246 76,861 75,195 38,383 11% 0.5x 0.5x 490 3,300/282 141,051 133,307 103,897 0% 0.7x 0.8x 1,688 6,620/40 682,228 No Data 483,053 0% 0.7x No Data 9,727 36,778/42 3,374,379 3,303,387 2,167,101 0% 0.6x 0.7x 14,712 112,356/90	613 764 524 5% 0.9x 0.7x 20 33/5 1 1,782 1,813 1,950 30% 1.1x 1.1x 4 56/55 0 2,608 3,094 2,160 16% 0.8x 0.7x 26 100/32 14 2,943 3,293 2,476 6% 0.8x 0.8x 46 100/19 14 8,842 8,440 5,591 19% 0.6x 0.7x 38 307/74 29 40,285 37,526 20,486 18% 0.5x 0.5x 81 1,435/246 9 76,861 75,195 38,383 11% 0.5x 0.5x 490 3,300/282 23 141,051 133,307 103,897 0% 0.7x 0.8x 1,688 6,620/40 52 682,228 No Data 483,053 0% 0.7x No Data 9,727 36,778/42 42 3,374,379 3,303,387	$\begin{array}{c ccccccccccccccccccccccccccccccccccc$

joins when combining posting list results from multiple full text and path value ranges.

The posting lists are not DML friendly structures. To keep up with high volume of DML rates, the index can be configured as synch on transaction commit or synch at certain intervals. It internally uses a log structured merge mechanism to initially keep less compacted posting lists tailored for fast DMLs and then gradually merge and compact multiple posting lists for the same key together into read friendly structures using parallel group by / aggregation query execution as the underlying map-reduce framework. As for functional indexes, OSON storage helps with indexing time as it can be more efficiently read in-place.

4.3.2 Sharding

NoSQL databases typically use sharding to scale horizontally in a distributed environment. Data is partitioned out to multiple machines based on a shard key which may be document id or document creation dates etc. Oracle Database also supports sharding stored JSON data, and each shard can have a locally defined JSON search index. NoSQL style OLTP and OLAP operations, with sharded key specifications, are routed to the corresponding shard server. NoSQL style cross-shard OLAP queries are distributed to each shard server to be executed in parallel. SQL/JSON operators, including JSON_TABLE(), are pushed down to each shard server.

5. PERFORMANCE EXPERIMENTS

We compare the performance of JSON text storage in BLOB with an 'IS JSON' check constraint to that of JSON type backed by OSON storage. Specifically, we show that document retrieval, ingestion, queries, and updates all benefit from using OSON. For JSON text storage, the document is stripped of insignificant whitespace and encoded as UTF-8 to be as small as possible. The database character set is ALU32UTF8. All experiments ran on a compute node in Oracle's US East (Ashburn) data center [17]. The compute node used was a VM.Standard2.24 [25] type virtual machine with the following characteristics: 24 OCPU/2.0 GHz Intel® Xeon® Platinum 8167M, 320 GB of memory, and 1TB encrypted block volume, ext4 [3]. In Section 5.2, we use two of these compute nodes - one for the database and one for the Java application. Otherwise, the experiments are run on a single node.

Figures presented in this section show ratios rather than absolute numbers. The purpose of these experiments is not to demonstrate the absolute performance numbers, but rather comparing the ratio among different approaches to show what relative improvement can be achieved. Also, since we use a variety of document sizes, using ratios helps show all the data in a single figure/scale. Measurements are done after a warm-up period so that the working set of data is fully loaded in-memory to eliminate measuring physical I/O overhead.

5.1 Encoding Size

We compare the encoding size of JSON text, BSON, and OSON. Encoding size is an important metric as it determines how much data is read and written to disk by the database as well as how much data is sent over the network to the client. We use a set of 11 JSON documents where 4 were taken from public sources and the rest were given to us by customers. Our goal here is to use only real-world, non-synthetic data that covers a variety of document sizes. Since we cannot disclose the specific customers and public sources used, we instead give each document a generic identifier (D1 - D11).

Table 1 summarizes the encoding sizes of the 11 documents we selected. The first column gives the identifier of the document while the next three columns give the size of the JSON text, BSON, and OSON values. The **Dict.** column gives the percentage of space taken by the OSON dictionary. The vsUTF8 and vsBSON columns give the size of JSON text and BSON relative to OSON. The last five columns give the total number of objects, keys, arrays, numbers, and strings in each document. The **#Keys** column gives two values - the total number of keys in the document followed by the total number of distinct keys.

In all cases except for D2, the size of the OSON encoding is smaller than the UTF-8, whitespace stripped JSON text. D2 is a small, flat document with no repeating keys, short key names, and small numeric values. In this case, the op codes and jump offsets in OSON made it 10% bigger than the corresponding JSON text. In all other cases, the OSON encoding is smaller than the corresponding JSON text, especially for documents having arrays of object structures or recursive structures that have many repeating keys. For example, the OSON encoding of D6 and D7 is half the size of the corresponding JSON text. D11 achieved the largest relative compression at one third the size of the JSON text. D11 is a large document with 1.8 million keys but only 73 distinct values. Thus, encoding these keys as numbers results in a large reduction in size. Similarly, the OSON encoding is typically significantly smaller than the corresponding BSON encoding for medium to large size documents because BSON does not encode key names as identifiers. For D9, the customer document

contained large numeric values that caused the 3rd party BSON encoder we used to produce an error and we report "No Data".

Just as JSON text and BSON, OSON is designed for OLTP use cases with OSON as an operating CRUD entity. Therefore, each OSON document is self-contained and it does not try to do common schema elements compression across document boundary as what relational storage would do by extracting and storing the common schema in central data dictionary. However, each OSON document contains common intra-document fields in the dictionary. Table 1 shows the percentage of the dictionary size relative to the whole OSON document in the Dict. column of Table 1. For small OSON documents, the OSON dictionary size represents average of 16% of the document. For medium to large OSON document, the OSON dictionary size is negligible. This means even if we do schema inference to extract and store common dictionary over a collection of OSON documents in a central dictionary, its results in 15% of saving in exchange for schema-evolution and migration issues. This is why the design of OSON using intra-document dictionary achieves the balance of schema flexibility and storage space compaction for OLTP JSON use case.

On the other hand, for OLAP JSON use cases, Oracle [28] is able to do common schema inference over JSON document collection to derive JSON Data Guide to shred JSON collection as relational storage with columnar in-memory format whose size is similar closer to that of Dremel [18] and STREED [27] style of columnar encoding. While these are efficient columnar encodings for OLAP queries through leaf scalar value dictionary encoding and SIMD scans [1,22], they lack of efficient random update capability at document level in comparison with OSON for efficient OLTP JSON use case in the same way as that of row store and columnar store of relational model for OLTP and OLAP use cases respectively.

5.2 Ingestion and Retrieval

5.2.1 Ingestion

In this experiment, we measure the insert throughput of a client Java application running on a compute node in the same data center as the database node. The application contains 24 threads and each thread starts with a pre-parsed, in-memory object model representation of the given document. Each thread repeatedly inserts the document into the database using a SQL insert statement. In the case of JSON text, the document object is converted to UTF-8 JSON text to be inserted into a BLOB column. In the case of OSON, the document object is converted to OSON binary at the client side and inserted into a JSON type column with server side OSON byte verification (section 4.1). After a warm-up period of 6 seconds, the average inserts per second is recorded over a 1-minute window.

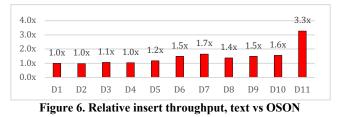


Figure 6 shows the insert throughput of JSON text compared to OSON binary storage. The figure gives the ratio rather than

absolute values since the insert rates for the smaller documents are much greater than that of the larger documents. For D1 through D4 the average writes per second is essentially the same showing that the cost of encoding OSON at client side is not a significant factor. Furthermore, for the remaining cases, the rate of insertion is significantly higher for OSON due to OSON storage of having a smaller size, reducing disk I/O and data transfer costs. For D11, the OSON is about a third of the size of the JSON text and we consequently get about three times the insert throughput.

5.2.2 Retrieval

In this experiment, we measure the read throughput of a client Java application running on a compute node in the same data center as the database node. The application contains 24 threads and each thread repeatedly retrieves the same document from the database and obtains a DOM API for the value. In the case of JSON text, we use a popular, 3rd party, JSON parser that we found to be the most performant of several other Java JSON parsers we tested. We use this parser to fully parse the JSON text and create an in-memory DOM for the value. In the case of OSON, we use DOM API implemented on top of our in-place client side OSON decoder.

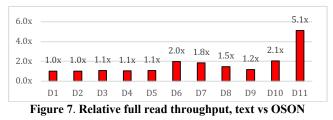


Figure 7 shows the relative read throughput when the client application fully iterates all of the values in the DOM. All JSON strings are accessed as Java strings and all numbers are converted into Java's BigDecimal. In this experiment, full document access performance with OSON is typically equivalent to or better than JSON text. For D11, the OSON is about a third of the size of the JSON text (saving on transfer costs) and reading the OSON is more efficient than JSON text parsing.

Figure 8 shows the relative read throughput when only a single client path is accessed on the client instead of iterating every value. For each document, we selected a path to some arbitrary value within the document (the path chosen is not shown). The specific path used is not important as both the JSON parsed object model using hash tables and the jump offsets within the OSON image will provide fast access to any point in the document. However, the advantage of OSON is that any point in the document can be read without pre-processing the rest of the document. In the case of JSON text, the entire document must be parsed before the path can be evaluated. This advantage is also useful to applications that need low latency access to the first value read within a document. For example, a page load on a social media feed can display the first 10 items without having to first process all 100 returned by the server. And, this can be achieved without any of the limitations or usability issues typically imposed by a streaming text parser. These benefits are most exaggerated for D11 where the relative increase is 7.7x as the client can skip reading portions of this large document.



Figure 8. Relative path read throughput, text vs OSON

5.3 Query Performance

In this section we compare query performance over JSON text and JSON type using 10 queries from the EYCSB [7] benchmark. In this experiment, there is no index created so that all queries use a table scan and evaluate the SQL/JSON operators over each document. The table used contains 2 million EYCSB JSON documents where the average size of a document is 2K. The total storage size of the table is 4.4 GB for JSON text and 4.2GB for OSON. The equivalent relational row storage by extracting all common meta-data and store them in central dictionary for EYCSB is 3.7 GB. Although OSON has 12% storage size overhead compared with pure relational row storage, in exchange for that, OSON storage has no schema evolution issue.



Figure 9. Relative response time, text vs OSON (no index)

Figure 9 shows a 6x-8x improvement in response time using OSON storage. SQL/JSON path expressions can be efficiently evaluated over OSON values in-place using the technique described in section 3.3.

As explained in section 4.2.3, JSON datatype is integrated with Oracle in-memory store by populating in-memory path/value indexes over OSON bytes. **Figure 10** shows an average of 200x response time improvement for the 10 EYCSB queries using IMC. Q6, Q7, and Q8 show a small improvement compared with the other queries because they return more rows that require post filters after in-memory index probes.

5.4 Update Performance

In this section we compare update performance of JSON type using JSON_TRANSFORM() for two customer datasets (D7 and D11 in Table 1). For JSON text storage, partial update is not feasible and, consequently, the corresponding transaction redo log size is proportional to the full document size. For OSON storage, transaction redo size is proportional to the actual delta update size, not the full document size. Consequently, partial update from OSON is significantly faster with significantly smaller redo size.

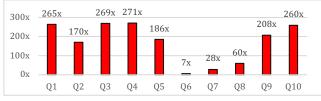


Figure 10. EYCSB response time, IMC index speedup

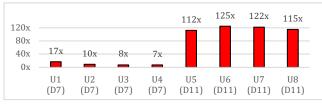


Figure 11. Update response time, full replacement versus piecewise update

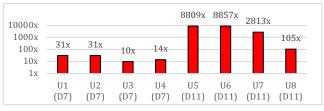


Figure 12. Update redo log size, full replacement versus piecewise update (log scale)

We used 4 statements (U1-U4) that use JSON_TRANSFORM() to update over 20,000 documents generated from D7. The total storage size is 1.5 GB for JSON text and 0.77GB for OSON. The equivalent relational row storage by extracting all common metadata and store them in central dictionary is 0.73 GB. So for large OSON document, its equivalent relational row storage has negligible storage size advantage.

For updating D7, Figure 11 shows 7x - 17x update response time improvement. Figure 12 shows 10x-31x transaction log redo size reduction. We used another 4 statements (U5-U8) that use JSON_TRANSFORM() over 40 documents generated from D11. The total storage size is 1.6 GB for JSON text and 0.55GB for OSON. For updating DOC11, Figure 11 shows 112x - 125x update performance improvement. Figure 12 shows 105x-8809xtransaction log redo size reduction.

6. RELATED WORK

Oracle [29], Microsoft SQL Server [10], MySQL [12], PosgreSQL [16], TeraData [23] support JSON text storage. MongoDB [11], TeraData [23], DB2 [6] support BSON storage. MySQL[12] and PosgreSQL[16] support their own binary storage. MySQL[12] and TeraData [23] support a JSON datatype. MySQL JSON datatype uses its own JSON binary format. TeraData supports JSON datatype with variety of storage formats: text, BSON [2], UBJSON [24]. Oracle supports fast in-memory OSON with JSON text storage on disk [28].

Similar to that of OSON [14], binary JSON formats from MySQL, PostgreSQL, Sinew[5], STREED [27] have all shared the common design strategy of supporting jump navigation based on indexed field key names in the binary format to speed up query. Therefore, all of these binary formats support faster query performance than JSON text storage. However, none of the query friendly binary formats addresses the issue of supporting partial update in the binary format. Our earlier work that used the OSON format loaded in-memory [28] does not address partial updates. The OSON format used for JSON datatype support in this paper does support partial update at binary format level. Mison [26] proposes SIMD based fast JSON parser for JSON query. However, it does not address partial update either. JSON text and streaming encoding based binary format, such as BSON [2] and UBJSON [24], are very limited to support partial update efficiently without replacing the full document. Compared with JSON_TRANSFORM() that combines multiple modification commands in one operator, MySQL[12] and PostreSQL[16] JSON update operator can only do a single update command at a time.

Sinew[5] and STREED [27] binary format separate dictionary meta-data out of binary formats to be stored in central location, this approach causes each binary instance to be non-selfcontained and hard to distribute without shipping central dictionary. Our design of OSON is self-contained without relying on a central location schema. Consequently, it is able to fully support database features such as partitioning, replication, sharding, import/export, and transportable tablespaces since it does not require any central schema synchronization.

Dremel [18] and STREED [27] columnar layout are examples of JSON columnar encoding formats. However, both assume that all JSON documents in a collection have the same schema. This is required to correctly re-assemble the original JSON document through columnar decomposition of the data. However, for a generic JSON datatype, we need to handle both homogeneous and heterogeneous JSON collections. We store the OLTP friendly OSON format on disk and, to achieve columnar scan performance, we leverage Oracle in-memory store [22] and IME expressions [1] to load columnar projections of JSON into memory for columnar based SIMD scan. Furthermore, we use path-value index and inverted keyword hash index format to facilitate JSON_EXISTS() and JSON_TEXTCONTAINS() in memory predicate evaluation and join.

SQL++ [9] (Couchbase) and the similar PartiQL (Amazon) [15] represent approaches that take a "JSON first" approach to extending a simpler SQL-92 language with a singular JSON-like data model and dynamic typing. Notably, these languages do not use explicit SQL/JSON operators or a secondary JSON path language to access JSON values. The Oracle simplified syntax for SQL/JSON (section 2.3) represents a somewhat similar approach but is defined on top of SQL-2016's SQL/JSON operators. SQL-2016 has been and will continue to be extended to support many more data models other than JSON. Oracle's approach integrates well with the latest version of the SQL standard and supports storing and querying many different data models side-by-side (relational, object relational, XML, JSON, spatial, graph, and so on). At the same time, it does not sacrifice the performance or expressiveness of queries over JSON data.

XML and JSON are both popular ways to represent semistructured data. XML is the more mature of these representations and has a larger body of academic and industry-led research on efficient processing using binary encodings. However, the JSON data model serves the same purpose and is semantically much simpler. XML is complicated by its document-oriented heritage, with concepts such as document node order, interleaved node types, mixed content, lack of in-line scalar data types, and no explicit array construct. In contrast, JSON doesn't specify a document node order, has explicit numeric and boolean types, and an explicit array type to connote ordering when it's required. The XML DOM interface requires implementations to provide costly semantics such as navigation to an explicit parent node and sibling traversals. XML has constructs such as attributes and namespaces that complicate processing without adding much expressive power for data representation. Our work on the OSON format has been able to exploit the simpler and cleaner JSON data model to allow

JSON DOMs to be persisted with minimal storage space, fast queries, and efficient updates that cannot be achieved with comparable binary XML DOM representations.

7. FUTURE WORK

Multi-value functional indexes over JSON documents will be supported since there can be multiple values in a JSON array within a JSON document. Multi-value functional indexes are used for JSON array range predicates in JSON_EXISTS() to determine a known set of scalar values that map to a given document. We will work with the SQL/JSON standards committee to add the JSON datatype, JSON_TRANSFORM(), and the simplified syntax and publish benchmark for hybrid JSON usecases.

8. CONCLUSIONS

The native JSON datatype, in combination with the SQL/JSON standard, establishes a strong semantic foundation to support JSON in a relational database. Using OSON as the encoding format enables efficient query, update, ingestion, and retrieval of JSON documents. Features such as SODA, full-text search indexes, and sharding have filled the main functional gaps with NoSQL databases. Further, we support high performance OLAP workloads over JSON by deeply integrating JSON type with Oracle in-memory, parallel query, materialized views, and storage cell smart scans. Combined, these features make a robust platform for processing JSON data. End-to-end OSON support accomplishes the goals of object databases - to be able to persist application level objects in a schema-flexible way without shredding them across multiple tables and rows.

9. ACKNOWLEDGEMENTS

The authors would like to thank Andrew Mendelsohn for his commitment of JSON datatype support, Tirthankar Lahiri, Juan Loaiza for support of JSON functionality as crucial feature in RDBMS, Cetin Ozbutun for his direction of in memory JSON support. Fred Zemke, Jan Michels for SQL/JSON standard. Martijn Cohen, Niraj Srivastava, Manoj Sahoo, Srinivas Kareenhalli for JSON/OSON performance evaluation. Sriram Krishnamurthy and his team: Rodrigo Fuentes, Jose Valera, Jose Alvizo, Sunitha Subramanyam, Sundeep Abraham, Hui Zhang for their help for implementation. Shasank Chavan, Aurosish Mishra, Saurabh Naresh Netravalkar for in-memory support. Kam Shergill for mem-optimized write support. Maxim Orgiyan and Tim Yu for SODA support. Venkat Venkatraman team for quality assurance. Andrew Witkowski, Tsae-Feng Yu for parallel query and Materialized view support. Rajendra Pingte, Saurabh Verma, Martha Woo Woo and their client teams to support JSON datatype access on all Oracle client tiers. Ron Decker for PL/SQL support. Wei Hu, Mark Dilman for sharding support. Yuhong Gu, Thuvan Hoang for replication support. Rich Phillips for DB import/export tool support.

10. REFERENCES

- A. Mishra, et al. Accelerating Analytics with Dynamic In-Memory Expressions. PVLDB 9(13), 1437-1448, 2016
- [2] BSON: http://bsonspec.org/
- [3] Block Volume:

https://docs.cloud.oracle.com/iaas/Content/Block/Concepts/overview.htm

- [4] Couchbase JSON Support: https://developer.couchbase.com/documentation/server/3.x/d eveloper/dev-guide-3.0/using-json-docs.html
- [5] D. Tahara, et al: Sinew: a SQL system for multi-structured data. SIGMOD Conference 2014: 815-826
- [6] DB2 JSON support: https://www.ibm.com/support/knowledgecenter/en/SSEPEK 11.0.0/json/src/tpc/db2z_jsonfunctions.html
- [7] EYCSB: https://blog.couchbase.com/ycsb-jsonbenchmarking-json-databases-by-extending-ycsb/
- [8] Elastic Search API: https://www.elastic.co/guide/en/elasticsearch/reference/1.4/in dex.html
- Kian et al: "The SQL++ Query Language: Configurable, Unifying and Semi-structured". http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.690 .8323
- [10] Microsoft SQL Server JSON support: https://docs.microsoft.com/en-us/sql/relationaldatabases/json/json-data-sql-server?view=sql-server-2017
- [11] MongoDB BSON and JSON : https://www.mongodb.com/json-and-bson
- [12] MySQL JSON DataType: https://dev.mysql.com/worklog/task/?id=8132
- [13] N. Mukherjee, et al. Oracle SecureFiles System. *PVLDB* 1(2), 1301-1312, 2008
- [14] Oracle Exadata Machine & Storage Server: http://www.oracle.com/us/products/database/exadata-techoverview-wp-1867437.pdf
- [15] PartiQL: https://partiql.org/
- [16] PostgreSQL with JSON and JSONB support: https://www.postgresql.org/docs/9.4/datatype-json.html

- [17] Regions and Availbility Domains https://docs.cloud.oracle.com/iaas/Content/General/Concepts /regions.htm
- [18] S. Melnik, et al: Dremel: Interactive Analysis of Web-Scale Datasets. VLDB 3(1), 330-339, 2010
- [19] SODA: Simple Oracle Document Access API: https://docs.oracle.com/en/database/oracle/simple-oracledocument-access/index.html
- [20] SQL NESTED Clause: https://docs.oracle.com/en/database/oracle/oracledatabase/19/adjsn/function-JSON_TABLE.html
- [21] SQL/JSON 2016 Standard: ISO/IEC TR 19075-6:2017, Information technology Part 6: SQL support for JavaScript Object Notation (JSON), http://standards.iso.org/ittf/PubliclyAvailableStandards/
- [22] T.Lahiri, et al.: Oracle Database In-Memory: A dual format in-memory database. ICDE 2015: 1253-1258
- [23] Teredata JSON Datatype: https://docs.teradata.com/reader/C8cVEJ54PO4~YXWXeX GvsA/4IAzgRsj_8aRj5pCQoEqzA
- [24] UBJSON: http://ubjson.org/
- [25] Virtual Machines: https://www.oracle.com/cloud/compute/virtualmachines.html
- [26] Y. Li, et al: Mison. A Fast JSON Parser for Data Analytics. PVLDB 10(10): 1118-1129, 2017
- [27] Z. Wang, et al: STEED: An Analytical Database System for TrEE-structured Data. PVLDB 10(12): 1897-1900, 2017
- [28] Z.H. Liu, et al. Closing the functional and Performance Gap between SQL and NoSQL. SIGMOD Conference 2016, 227-238
- [29] Z.H. Liu, et al. JSON data management: supporting schemaless development in RDBMS. SIGMOD Conference 2014, 1247-1258 2014