Closing the functional and Performance Gap between SQL and NoSQL

Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Ying Lu, Hui Joe Chang

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065, USA

{zhen.liu, beda.hammerschmidt, doug.mcmahon, ying.lu, hui.x.zhang}@oracle.com

ABSTRACT

Oracle release 12cR1 supports JSON data management that enables users to store, index and query JSON data along with relational data. The integration of the JSON data model into the RDBMS allows a new paradigm of data management where data is storable, indexable and queryable without upfront schema definition. We call this new paradigm **Flexible Schema Data Management** (FSDM). In this paper, we present enhancements to Oracle's JSON data management in the upcoming 12cR2 release. We present **JSON DataGuide,** an auto-computed dynamic soft schema for JSON collections that closes the functional gap between the fixed-schema SQL world and the schema-less NoSQL world. We present a self-contained query friendly binary format for encoding JSON (**OSON**) to close the query performance gap between schema-encoded relational data and schema free JSON textual data. The addition of these new features makes the Oracle RDBMS well suited to both fixed-schema SQL and flexible-schema NoSQL use cases, and allows users to freely mix the two paradigms in a single data management system.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Relational databases, transaction processing.*

General Terms

Algorithms, Management, Performance, Design, Standardization, Languages.

Keywords

JSON, SQL/JSON, Schema-less, Flexible Schema, NoSQL, XML, SQL/XML.

1. INTRODUCTION

Using a 'schema first, data later' approach, RDBMS platforms are very successful at managing well-structured relational data. NoSQL systems [4] are challenging this with a 'data first, schema later or never' paradigm. The fixed-schema paradigm of relational data management, which demands a schema before data can be stored and queried, is at odds with the NoSQL world where a variety of data in continually changing forms is available from diverse data sources. To address these challenges, the Oracle 12cR1 release supports JSON as a simple and practical example of FSDM to unleash the power of schema-less data management [21]. It is based on following three engineering principals originating from ORDBMS [11, 13]:

- Storing JSON objects as aggregated, self-described entities without shredding them into relational rows and columns. That is, embracing the document-object storage model. This resolves the "birth pain" usability problem of the schema-rigid classical RDBMS SQL world [8].
- Indexing JSON using a schema-agnostic strategy to support ad-hoc queries that search both schema and values together. That is, extending inverted index technology to index both data and schema together [18]. This resolves the problem of requiring knowledge of the query workload before an index can be defined in fixed-schema systems.
- Querying JSON using SQL as the inter-document query language and SQL/JSON path language as the intra-document query language [21]. This avoids the problem of creating a brand-new set-query language, such as JSONiq [12], to query JSON only.

These three principals have enabled FSDM [32] in RDBMS. However, we don't think that the relational model and SQL are out-dated [1] due to the demand for FSDM. The strength of the relational model [5] is that it avoids imposing a single hierarchy to manage data, and the strength of SQL is its ability to deliver powerful data analytics with its set-based declarative query language. These strengths motivate us to integrate FSDM capabilities with the relational model and with SQL. Our key insight is this: rather than integrate the two worlds by trying to impose a fixed relational schema into which FSD would be decomposed, we integrate them by dynamically projecting a relational schema **continuously derived** from collections of FSD instances. This **logical dynamic schema** allows users to view and query FSD relationally. *That is, a relational schema is not constructed as a physical frame to fit the data but rather as a logical lens to view the data*. We don't use schema to encode and store data, so FSD physical storage is free from the need for schema evolution.

To realize this vision, a relational view over JSON data can be defined using the SQL/JSON query construct JSON_TABLE() as a virtual function to derive relational rows from JSON data [21]. Once JSON relational views are defined, users can write queries on top of the relational views to achieve compile time schema check with the rich analytic power of SQL fully applied to JSON data. To define such JSON_TABLE() views, users would need to figure out the implied schema within a collection of JSON instances so that they can write the SQL/JSON paths needed by the JSON_TABLE() operator. For relational data, the problem of deriving the relational schema from the data and then registering the relational schema with the RDBMS is traditionally accepted by the developers of the relational application. In FSDM, the schema derivation problem is the responsibility of the underlying DBMS. The "birth pain" [8] experienced by users in fixed-schema systems is resolved by having an FSD enabled RDBMS that *automatically computes and maintains the implied schema for an FSD collection and assists users in defining relational views to allow SQL access to the data.*

The computed schema must be *dynamic*: it must continually evolve as instances are added to or changed within FSD collections. For a JSON collection, the automatically computed schema should include a derivation of all the hierarchical structural paths that exist in that JSON collection, as well as the data types and statistics of leaf scalar values. We call this automatically-maintained schema the **JSON DataGuide**. We support two forms of the JSON DataGuide: persistent and transient. The persistent DataGuide is provided as a component of a schema agnostic JSON search index [21]. The transient schema is provided as a new SQL aggregate function over any JSON collections that can be computed from SQL declaratively. The concept of DataGuide has introduced a powerful new paradigm: "*write without schema, read with schema*". That is, users can store data without providing a schema definition, but they can still query the data as if a schema were defined and registered to the system ahead of storing the data. This is the key value proposition for the FSD enabled RDBMS.

The RDBMS uses relational schema information to encode data for storage and to decode data for query. The row format is optimized for fast query performance. To achieve query performance and minimize the storage size, row data is not self-describing; an external schema describing column meta-data is necessary to interpret the columns within the data. An FSD enabled RDBMS is **not** able to rely on column schema to encode data. Therefore, it needs an efficient binary format to encode data for fast query performance without relying on the existence of a central schema for column meta-data. For JSON, we have developed a binary format, called **OSON** (Oracle binary JSON encoding), to meet these requirements. OSON is a self-contained, compact representation of a tree of structures, arrays, and scalars. It's designed to provide rapid navigation to elements of the tree to resolve SQL/JSON path expressions. OSON bytes can be encoded from JSON text [10], BSON [16] or AVRO [15], and may either be stored persistently or loaded and used as structures in memory by Oracle's in-memory database option[19] for fast SQL/JSON query processing.

The main contributions of this paper are:

- To bridge the functional gap between the fixed-schema world of SQL and the schema-less NoSQL world, we introduce the concept of a DataGuide, a logical dynamic soft schema in RDBMS. We describe a practical SQL based solution to compute and maintain the DataGuide declaratively and automatically for both persistent and transient JSON collections so that they can be accessed relationally. The key idea here is that in an FSD enabled RDBMS, schema is no longer static meta-data but rather dynamically and continuously derivable via SQL over FSD collections.

- To bridge the performance gap between fixed-schema encoding of relational data and schema free JSON textual data, we describe a self-contained, compact binary encoding of JSON (OSON) which can be an order of magnitude faster than textual JSON for supporting SQL/JSON queries. The OSON format needs no central schema for column meta-data but still gives column data access performance that is close to a schema encoded row format.

- To bridge the conceptual gap between SQL and NoSQL world, we practically integrate the idea of '**Schema-less for Write**' from NoSQL [4] world and '**Schema-Rich for declarative Query**' from SQL world [5] through management of JSON data in a single RDBMS product without the need of polyglot persistence [7] . 'Schema-less for write' resolves the schema birth pain problem [8] and continuous schema evolution problem associated with classical RDBMS. 'Schema rich for declarative query' resolves the issue of developing code to discover data schema and to procedurally query data associated with pure NoSQL data stores. Figure 1 shows our architecture of connecting SQL and NoSQL world on top of a single RDBMS platform with SQL remains as declarative set query language.

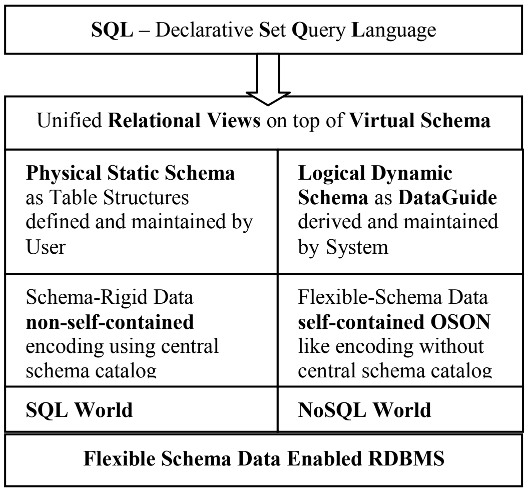
SQL – Declarative Set Query Language
Unified Relational Views on top of Virtual Schema
Physical Static Schema as Table Structures defined and maintained by User
Logical Dynamic Schema as DataGuide derived and maintained by System
Schema-Rigid Data non-self-contained encoding using central schema catalog
Flexible-Schema Data self-contained OSON like encoding without central schema catalog
SQL World
NoSQL World
Flexible Schema Data Enabled RDBMS

Figure 1 – Architecture connecting SQL and NoSQL World

Outline: Section 2 compares our work to related work. Section 3 describes the JSON DataGuide in detail. Section 4 describes the OSON design in detail. Section 5 describes SQL/JSON performance enhancements using OSON. Section 6 presents an evaluation of performance. Section 7 discusses future work. Section 8 concludes the paper with acknowledgements.

2. RELATED WORK

NoSQL vendors [4] have used JSON as a simple semi-structured data format for the document object model. RDBMS vendors are integrating support for JSON data with their SQL engines, including IBM DB2 [29], Microsoft SQL Server [25], TeraData [26], Vertica [27] and Oracle 12cR1 [21]. Another popular approach is to add a level of abstraction over polyglot persistence in middle-tier code. One example of this approach is Sinew [22]. However, Oracle 12cR2 takes this a step further. Recognizing

that the lack of a schema presents a challenge in schema-less NoSQL world, Oracle offers the JSON DataGuide, allowing users to write queries over JSON collections with a de facto schema continuously derived from the instance documents. Concept of DataGuide was proposed by researchers in the LORE system [28]. Dynamic discovery and extraction of schema from a document store was presented in [20]. However, both these prior efforts dealt only with pure document store database, not in SQL based RDBMS. Our unique contribution is to deeply integrate DataGuide as a dynamic schema facility with existing SQL mechanisms: view, virtual column, index and aggregation. Our JSON DataGuide service provides automatic and customizable generation of SQL/JSON JSON_TABLE() based relational views for nested hierarchical documents, and virtual columns for singleton scalar values. While the flexible table approach of Vertica [40] is limited to singleton scalar fields from JSON collection, our approach handles JSON array expansion gracefully using the *NESTED PATH*- construct of JSON_TABLE(). That is, the contents of JSON arrays are un-nested automatically and become directly available to SQL queries as simple columns.
Unlike approach in Sinew [22], our JSON DataGuide is fully integrated with the Oracle RDBMS kernel and the SQL language. We provide support for both persistent and transient dynamic schemas. The *persistent JSON DataGuide* is integrated as a component of a schema agnostic JSON search index, ensuring that both discovery and search of JSON structures are completely in synch. Using a single index, users can discover JSON path structures within a collection, as well as what documents within the collection have particular path structures and values. The *transient JSON Dataguide* is provided as a SQL aggregation function. With this, computing an ad-hoc JSON DataGuide is simple, flexible, and declarative, and can be accomplished in one SQL statement!
There are number of popular binary encoding formats for hierarchical data objects, among them BSON [16], AVRO [15], and Protocol Buffers [14]. None of these is fast enough for all query situations, because the serialization formats lack efficient random field access, as discussed in Sinew[22]. Like Sinew [22], OSON encoding supports efficient random field access. Unlike Sinew [22], the OSON format is self-contained and doesn't rely on an external central catalog system to manage attribute ids. We have found that depending on such an external system is tantamount to maintaining a schema: exactly what users are trying to avoid with these NoSQL stores. The OSON format supports arbitrary SQL/JSON path navigation query efficiently. The self-contained format makes OSON as portable as JSON text or BSON binaries while still providing rapid random tree navigation; in contrast, BSON and textual JSON must be accessed in a serial fashion, with at best the ability to perform skip navigation. Although tree-encoding for XML is done in DB2 XML storage format [2], our OSON design is compact tree format: well-suited both as a persistent format and as the basis for an in-memory format. Our OSON and virtual column query access is tightly integrated with Oracle in-memory database architecture [19]. To the best of our knowledge, this is the first industrial paper that provides comprehensive SQL solution for providing dynamic schema support for JSON and in-memory query processing for JSON and thus towards the direction of closing schema and performance gap between SQL world and NoSQL world.
The comparison of OSON format and Dremel [24] representation is discussed in section 7.

3. JSON DataGuide

3.1 JSON DataGuide Conceptual Description

JSON is a language-neutral representation of data structures and scalar types common across different programming languages. The JSON data model is hierarchical and can be regarded as consisting of three types of nodes arranged in a node tree. The three node types are JSON objects, JSON arrays, and JSON scalars. JSON objects are structures consisting of key/value pairs with string keys and node values. JSON arrays are ordered lists of node values. JSON objects and JSON arrays are both considered container nodes. JSON scalars are always leaf values and may be strings, numbers, booleans, or null. A JSON document can be parsed and constructed as a JSON DOM tree using the JSON data model. SQL/JSON path language semantics [21] are based on the JSON DOM model.
A JSON DataGuide for a single JSON document instance is computed by extracting the container node *skeleton* of the JSON DOM tree. Leaf scalar values are replaced by data type and length. A JSON DataGuide for a collection of JSON documents is simply a merge of the instance DataGuides across all documents in the collection. The merge union process removes duplicate tree paths if they have the same tree node type for each step in the path. Paths having different tree node types at any step are considered different. Leaf scalar data information is merged by replacing conflicting data types with a more general type, and using the maximum length.
As an example of path merge, in one JSON document, path *'$.a.b'* with node *"b"* is a scalar tree node type, while in another JSON document, path *'$.a.b'* with node *"b"* is an object node type. The merged JSON DataGuide keeps both paths: *'$.a.b'* as a scalar node type and as an object node type, respectively. As an example of merging scalar nodes, in one JSON document, path *'$.a.b'* is a number, while in another JSON document, path *'$.a.b'* is a string. The resulting JSON DataGuide keeps one structural path *'$.a.b'* as scalar tree node type with *string* as leaf data type.
The JSON collection is stored in a column with an 'IS JSON' check constraint created. For such a persistent JSON column, we support a persistent JSON DataGuide that is incrementally maintained as new JSON documents are inserted into the collection.

3.2 Persistent JSON DataGuide Maintenance

3.2.1 DataGuide Evolution with DML

Consider a JSON column **JDOC** of a table **PO** storing the following purchaseOrder JSON documents. There is a nested detail hierarchy *"items"* under the master *"purchaseorder"* in each document.

DID	JDOC
1	*{"purchaseOrder":{"id" : 1, "podate" : "2014-09-08", "items" : [{"name":"phone" , "price" : 100, "quantity" : 2}, {"name":"ipad", "price" : 350.86, "quantity" : 3}]}}*
2	*{"purchaseOrder":{"id" : 2, "podate" : "2015-03-04", "items" : [{"name":"table" , "price" : 52.78, "quantity" : 2}, {"name":"chair", "price" : 35.24, "quantity" : 4}]}}*

Table 1 – JSON PurchaseOrder Collection

A persistent JSON DataGuide is maintained as a component of the JSON search index. The JSON search index is a general purpose schema agnostic index created on a JSON column by maintaining an inverted index for every JSON field name and every leaf scalar value (strings are tokenized into a set of keywords to support full-text searches). The index allows ad-hoc SQL/JSON path query predicates such as *JSON_EXISTS()*, *JSON_TEXTCONTAINS()*, and *JSON_VALUE()* to be evaluated efficiently [21] over the JSON collection. The JSON search index is the most natural place to support JSON DataGuide maintenance because this index is incrementally maintained when documents in the index JSON column are added, removed, or replaced. *To minimize the overhead, the DataGuide maintenance is incorporated directly into the processing of the IS JSON check constraint.* In the common case where a new JSON instance doesn't result in any new path structures or scalar node changes, the DataGuide processing terminates without the need to call any persistent DataGuide processing module.

The JSON search index internally stores the persistent JSON DataGuide in a relational table $DG. Using the data from Table 1 as an example, the contents of the $DG table are shown in Table 2. The $DG table captures all the distinct paths in the JSON document collection with its leaf type.

Path	Type
$.purchaseOrder	object
$.purchaseOrder.id	number
$.purchaseOrder.podate	string
$.purchaseOrder.items	array
$.purchaseOrder.items.name	array of string
$.purchaseOrder.items.price	array of number
$.purchaseOrder.items.quantity	array of number

Table 2 – JSON DataGuide Relational Format in $DG

To show how the DataGuide evolves in the face of new instances, consider what happens when the document shown in Table 3 is added to the collection. Note a new child object *"parts"* is added below the existing "*items*" array – this causes the DataGuide to grow **deeper**. A new top-level *"foreign_id"* field is also added.

DID	JDOC
3	*{"purhcaseOrder":{"id" : 2, "podate" : "2015-06-03",* ***"foreign_id" : "CDEG35",*** *"items" : [{"name":"TV" , "price" : 345.55, "quantity" : 1,* ***"parts" : [{"partName" : "remoteCon", "partQuantity" : "1"}, {"partName" : "antenna", "partQuantity" : "2"}]*** *}, {"name":"PC", "price" : 546.78, "quantity" : 10,* ***"parts" : [{"partName" : "mouse", "partQuantity" : "2"}, {"partName" : "keyboard", "partQuantity" : "1"},]*** *}]}}*

Table 3 – introducing a new child hierarchy

JSON search index maintenance inserts 4 new rows into $DG (see Table 4) to reflect the new path structures encountered for the document shown in Table 3.

Path	Type
$.purchaseOrder.items.parts	array of array
$.purchaseOrder.items.parts.partName	array of string
$.purchaseOrder.items.parts.partQuantity	array of string
$.purchaseOrder.foreign_id	string

Table 4 – four new rows added into $DG

Now consider what happens when the document show in Table 5 is added to the collection. A new hierarchy *"discount_items"* is added as a sibling of existing node "*items*" – this causes the DataGuide to grow **wider**.

DID	JDOC
4	*{"purchaseOrder":{"id" : 98, "podate" : "2015-07-04", "items" : [{"name":"CD" , "price" : 5.55, "quantity" : 10, }, {"name":"DVD", "price" : 6.78, "quantity" : 20, }],* ***"discount_items": [{"dis_itemName" : "CPH", "dis_itemPrice" : 105.52, "dis_itemQuanitty":2, "dis_parts" : [{"dis_partName" : "phonejack", "dis_partQuantity" : 3}, {"dis_partName" : "plug", "dis_partQuantity" : 2},] }, {"dis_itemName" : "Printer", "dis_itemPrice" : 121.33, "dis_itemQuanitty":9, "dis_parts" : [{"dis_partName" : "toner", "dis_partQuantity" : 5}, {"dis_partName" : "paper", "dis_partQuantity" : 20},] }]*** *}}*

Table 5 - introducing a new sibling hierarchy

JSON search index maintenance inserts 7 new rows into $DG (see Table 6) to reflect the new path structures encountered for the document shown in Table 5.

Path	Type
$.purchaseOrder. discount_items	array of array
$.purchaseOrder. discount_items.dis_parts	array of array
$.purchaseorder. discount_items.dis_parts.dis_partName	array of string
$.purchaseOrder. discount_items.dis_parts.dis_partQuantity	array of number
$.purchaseOrder.discount_items.dis_itemName	array of string
$.purchaseOrder.discount_items.dis_itemPrice	array of number
$.purchaseOrder.discount_items.dis_itemQuanitty	array of number

Table 6 - seven new rows added into $DG

In addition to storing path and type information, the $DG table also has columns that store statistical information for a path such as frequency, minimum and maximum values, and number of null values. These statistical columns are populated when the JSON search index statistics are computed.

3.2.2 DataGuide Representation

The Persistent dataGuide can be presented in two forms: a *flatten form* as what is stored in $DG table and a *hierarchical form* with nested structures. Both forms are encoded as a JSON document that can be returned as a CLOB by invoking a PL/SQL function *getDataGuide()* from persistent indexing layer. In particular, this function can aggregate the information from the $DG table into a hierarchical format represented as a single JSON document. Users can annotate the computed DataGuide by picking fields, re-naming column names, changing the maximum length of data types, etc., and then call *CreateViewOnPath()* with the annotated DataGuide to generate customized relational views discussed in section 3.3.2.

3.3 Virtual Relational Schema for JSON

To realize the benefit of *'write without schema, read with schema'*, Oracle provides PL/SQL procedures to assist users in projecting relational views and virtual columns of the data, driven by information from the JSON DataGuide. The DataGuide is used as a dynamic schema to automatically compose SQL/JSON operators, making JSON data appear as if it were physically shredded and stored in master detail relational tables.

3.3.1 Adding Virtual Columns (VC) using JSON_VALUE()

With JSON DataGuide computed, user can run a PL/SQL procedure *AddVC()* to automatically add virtual columns to the base table. Each virtual column is defined using JSON_VALUE() to project a singleton scalar value out of the JSON document. Using the JSON DataGuide presented in section 3.2 as a starting point, Table 7 shows the definition of 3 virtual columns that are added to PO table via *AddVC()* calls. These virtual columns can be referenced in a query as if these fields in base JSON document were physically shredded and stored in them.

VC Name	VC
JCOL$id	*JSON_VALUE(JCOL,'$.purchaseOrder.id returning number)*
JCOL$podate	*JSON_VALUE(JCOL,'$.purchaseOrder.podate' returning varchar(16))*
JCOL$foreign_id	*JSON_VALUE(JCOL,'$.purchaseOrder.foreign_id returning varchar(8))*

Table 7 – JSON_VALUE() Virtual Columns

3.3.2 Creating De-normalized Master-Detail Views (DMDV) using JSON_TABLE()

Virtual columns can only project singleton scalar values; that is, values having a one-to-one relationship with document instances. Often, however, scalar values relevant to queries are nested within arrays and have a many-to-one relationship with document instances. If shredded into relational tables, the most natural way to view them is to form a left outer join of master record to detail records so that fields from a master record are repeated for each detail record. The outer join is used instead of an inner join to ensure that master records are captured in the view even if there are no matching detail records. In cases where there are more than one sibling detail arrays to be joined for the same master record, fields from the sibling records can be exposed using a union join. A union join is equivalent to a full outer join with an impossible condition, such as 0=1. By design, the default of the JSON_TABLE() *NESTED PATH* clause is to un-nest JSON arrays with left-outer join semantics for child hierarchies and do a union join for sibling hierarchies. With JSON DataGuide computed, user can run a PL/SQL procedure *CreateViewOnPath()* to automatically create a JSON_TABLE() view. We call such views as De-normalized Master Detail Views (DMDV) because the output is the same as the output of a view over physically decomposed master detail tables using the outer join and union constructs that we just described. DMDV is structurally similar to the wide table described in [30]. Table 8 shows the definition of a DMDV generated by calling *CreateViewOnPath('$')* . Users can also generate a DMDV view for a particular path. For example, users can generate a DMDV view for the items detail branch alone by executing *CreateViewOnPath('$.purchaseOrder.items')*. If JSON index statistics are collected, then frequency information can be passed to *CreateViewOnPath()* to project fields only if they occur more frequently than a given threshold value. In this way, sparse and outlier fields can be eliminated as columns of the DMDV.

```
CREATE VIEW PO_RV AS
SELECT PO.DID, JT.*
FROM PO, JSON_TABLE("JCOL" FORMAT JSON, '$'
COLUMNS
 "JCOL$id" number path '$.purchaseOrder.id',
 "JCOL$podate" varchar2(16) path '$.purchaseOrder.podate',
 "JCOL$foreign_id" varchar2(8) path '$.purchaseOrder.foreign_id',
 NESTED PATH '$.purchaseOrder.items[*]' /*NP1*/
  COLUMNS (
   "JCOL$name" varchar2(8) path '$.name',
   "JCOL$price" number path '$.price',
   "JCOL$quantity" number path '$.quantity'),
  NESTED PATH '$.parts[*]'  /*NP2*/
    COLUMNS ("JCOL$partName" varchar2(16) path '$.partName',
                  "JCOL$part Quantity" varchar2(1) path '$.partQuantity'),
 NESTED PATH '$.purhcaseorder.discount_items[*]'  /*NP3*/
   COLUMNS (
    NESTED PATH '$.dis_parts[*]'  /*NP4*/
      COLUMNS ("JCOL$dis_partName" varchar2(16) path '$.dis_partName',
                    "JCOL$dis_partQuantity" number path '$.dis_partQuantity'),
                    "JCOL$dis_itemName" varchar2(8) path '$.dis_itemName',
                   "JCOL$dis_itemPrice" number path '$.dis_itemPrice',
                   "JCOL$dis_itemQuanitty" number path '$.dis_itemQuanitty') ) JT
/* NP1 & NP2 are left outer join, so do NP3 & NP4; NP1 & NP3 are union join */
```

Table 8 – JSON_TABLE() DMDV View

3.4 Transient JSON DataGuide Computation

Note, persistent JSON DataGuide is additive, it does not remove paths for documents that are deleted. However, a JSON DataGuide can be computed dynamically by executing a SQL aggregation function *JSON_DataGuideAgg()* over the result of any SQL query returning a set of JSON documents. JSON_DATAGuideAgg() is implemented using user defined aggregation framework from ORDBMS [11,13]. It computes and returns JSON DataGuide as a single JSON document in either flatten form or hierarchical form as discussed in section 3.2.2. Table 9 shows SQL queries that compute transient JSON DataGuide. Q1 computes the DataGuide by sampling 50% of JSON documents in a JSON collection. Q2 computes the DataGuide group by their insertion date. Q3 computes the DataGuide for a filtered subset of JSON documents that have JSON path *'$.purchaseOrder.foreign_id'*. Integrating JSON DataGuide as a SQL aggregation function provides declarative way to compute DataGuide for JSON collection that may not be stored in Oracle. For example, Oracle external table can map file system data as virtual relational table on top of which JSON DataGuide can be computed and DMDV view can be created for query. Oracle SQL/JSON query support can transparently read from external virtual table and thus enables the In-Situ Query processing over JSON collection.

Q1	*Select json_dataguideagg(jcol) from po sample (50)*
Q2	*Select json_dataguideagg(jcol) from po group by insertion_date*
Q3	*Select json_dataguideagg(jcol) from po where json_exists(jcol, '$.purchaseOrder.foreign_id')*

Table 9 – Example queries computing JSON DataGuide Transiently

4. JSON Binary Format: OSON

4.1 OSON Design Criteria

In the classic relational approach to data management, schema information is separated from instance data, with the schema being defined in a central dictionary and then used to encode data instances for storage. In contrast, flexible storage models typically store metadata together with data instances, such that instances can be decoded and understood without the need to refer to a central dictionary. Self-contained JSON document instances can be more easily distributed, replicated, imported, exported, evolved, without the need for costly synchronization. *This property of self-containment is the first design criteria for our binary JSON format.*

BSON [16] is self-contained and is readable without costly textual parsing. The BSON format has a symmetric structure for nested child containers. Because of this, child containers can be extracted

or inserted by simply copying byte ranges, a potentially big advantage when doing projections. Because containers have leading length words, a BSON decoder can skip unneeded child containers without decoding all of their descendants. The cost of nested symmetry is that the BSON format needs to store field names at each object level, and must repeat them for elements within arrays of objects. JSON path evaluations involving field name searches require relatively expensive string comparisons. A lesser problem is that field names are null terminated, necessitating a byte scan for the end of the field name string during searches. Finally, BSON, like JSON, is a stream format potentially requiring that an entire instance be read to resolve a JSON path, or, worse, to determine that it's not present in the instance. *Enabling readers to jump inexpensively to named fields and array locations is our second design criteria.*
JSON scalars can be numbers, strings, or Booleans. In binary formats this set is commonly extended to include date, timestamp, and raw data types. The SQL/JSON path language and SQL/JSON operators such as JSON_TABLE() support type-aware operations, such as type-based comparison semantics and arithmetic expressions. In a binary format, scalars can be stored in a format native to the database engine for maximum performance, especially when values pass between the JSON and SQL worlds. *Encoding scalar values in the same binary format as our SQL scalar columns is our third design criteria.*

4.2 OSON Format: Three-Segment Architecture

The OSON format separates a JSON document into 3 segments: a field-id-name dictionary segment, a tree node navigation segment and a leaf scalar value segment with navigation segment containing references to dictionary and value segment as shown in Figure 2. This division is done to reflect the different loading and access properties.

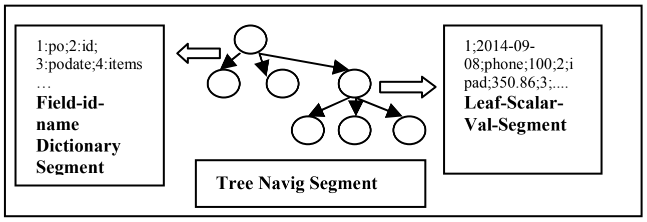
1:po;2:id;
3:podate;4:items
…
Field-id-name Dictionary Segment
Tree Navig Segment
1;2014-09-08;phone;100;2;ipad;350.86;3;....
Leaf-Scalar-Val-Segment

Figure 2 – OSON Format Architecture

4.2.1 Field-id-name-dictionary segment

Each unique field name within a JSON document is assigned an integer field name identifier and is stored only once so that multiple references to the same field name are replaced with its field name identifier. Because the identifiers take just a few bytes, the OSON encoding typically saves significant space when the original document has a nested array of sub-objects, or encodes a recursive hierarchical structure. *The biggest benefit of assigning identifiers is that they can facilitate rapid navigation to a child field given by performing a binary search using integer comparisons.* OSON stores the field name identifiers for each object in **sorted** order to support this type of access.
The field-id-name dictionary segment is responsible for providing a fast dictionary mapping between a field name and a field name identifier. The assignment of a field name identifier to a field name is done arbitrary using a hash function. The OSON encoder applies the hash function to each distinct field name referenced in a JSON document and builds a hash table containing all such mappings for the document. The hash table is compacted into a hash-id-array. Each array entry stores the field name and the hash id of the field name. The entire hash-id-array is **sorted** by the hash ids. *The ordinal position of an array entry containing a field name is used as the field name identifier for that field name.* To look up a field name identifier given a field name, we first apply the hash function to the field name to get its hash id. Then we perform a binary search on the hash-id-array using the hash id to locate the array entries having that hash id. Finally, we perform a field name string match to resolve any possible hash collisions.
To avoid repeatedly calling the hash function on field names and to avoid field name identifier lookup altogether, we have applied the following optimizations to the SQL/JSON query compilation and execution time.

- During the SQL query compilation phase, all SQL/JSON paths are compiled. The same hash function is called on all distinct field name references in all SQL/JSON paths so that their corresponding hash ids are stored in the SQL execution query plan.
- During SQL execution time, as the SQL/JSON path query is applied to each OSON instance, we use the pre-computed hash id and the field name that are stored in the execution plan to find the field name identifier using the field-id-name dictionary segment. This is done once for the OSON instance; afterwards, all field name searches are efficiently executed as binary searches using the instance-specific field name identifier.
- Finally, when the SQL/JSON path query is applied to the next OSON instance, the mapping determined on the previous OSON instance is checked to see if it matches the next instance. This single-row look-back is very effective on collections of structurally homogeneous JSON document instances, because the same field name identifiers are most likely used repeatedly, and therefore the cost of field-id resolution can often be skipped.

4.2.2 Tree-node Navigation Segment

The tree node navigation segment of OSON represents the JSON document as a tree-like skeleton of nodes that supports navigation from a parent node to its child nodes. There are 3 types of JSON tree nodes: JSON object nodes, JSON array nodes, and JSON scalar nodes. Each node is identified by its byte offset location from the beginning of the tree-node navigation segment. The byte offset is used as the tree node address and is stored in the child array of the parent tree node to jump to the child tree node. Each node has a common tree node header byte storing the type of the node (Object, Array, Scalar), and property flags that vary depending on the type of the node.

- For a JSON object node, the child array stores all of its children's field name identifiers and their corresponding tree node addresses. The array is sorted by the field name identifiers to support rapid navigation to the child node given its field name.
- For a JSON array node, the child array stores all of its children's node addresses. The child in Nth array position is accessed by taking the child node's offset from the Nth position in the array.
- For a JSON scalar node, this records the data type of the scalar value and an offset into the leaf-scalar-value segment where the data bytes are located. JSON boolean and null values are directly encoded as bit flags in the JSON scalar node and therefore have no offset. For fixed length scalar values, the length is inferred from the data type. For variable

length scalar values, bit flags are used to indicate if the length is encoded using one, two, or four bytes.

4.2.3 Leaf-scalar-value segment

The leaf-scalar-value segment is comprised of all leaf values in a JSON object instance. The bytes representing the values are concatenated together to form this segment. Each leaf scalar value is found by its byte offset from the beginning of the segment. The byte offset is used as the leaf value address and is stored in the JSON scalar node. For variable length data items, a length is stored before its value. For fixed length data, no length is stored. By default, OSON uses the Oracle binary number format to encode JSON numbers, minimizing the cost of using these values in SQL. JSON numbers can also be encoded using IEEE double-precision format, or encoded as strings in a canonical format.

No binary format is optimally efficient in its support for queries and updates. In designing OSON, we opted to maximize path query efficiency, so partial update support is limited to changes of existing leaf scalar values. However, we will work on enhancing OSON format to make intelligent trade-off to be more update friendly to handle future SQL/JSON partial object update language requirement.

5. SQL/JSON Query Performance Improvement

5.1 JSON DOM Path Engine using OSON based JSON DOM implementation

To evaluate SQL/JSON operators on textual JSON documents, we developed a JSON path engine that operates in a streaming fashion, using a series of *events* produced by the JSON text parser. These events allow evaluation of SQL/JSON operators without the need to materialize JSON objects in memory as document object model (DOM) instances. Generating the events requires costly text parsing. Worse still, evaluation of more complex operators requires the engine to memorize event sequences, in effect partially or completely negating the benefit of avoiding DOM construction. These operators include the JSON_TABLE() virtual table [21] as well as complex JSON_EXISTS predicates. In many cases it’s simpler and more efficient to evaluate SQL/JSON operators on a DOM tree instance. The OSON format allows DOM read operations directly against the serialized instance, avoiding the need for a costly parse and DOM construction. Each JSON DOM tree node address is an offset within the tree-node-navigation segment of OSON, so DOM read operations can jump directly to desired nodes using the offsets in lieu of the machine pointer dereferences. This allowed us to extend the path engine so that it is capable of operating directly on DOM instances, without sacrificing efficiency, and without the need for memorizing events.

The DOM version of the JSON path engine calls the following interfaces to evaluate each step of a JSON path step and apply any predicate expression underneath it:

- *JsonDomGetNodeType(treeNodeAddress)*
- *JsonDomGetFieldValue(treeNodeAddress, fieldNameIdentifier)*
- *JsonDomGetArrayElement(treeNodeAddress, arrayIndexRange)*
- *JsonDomGetScalarInfo(treeNodeAddress)*

For path steps that require finding a child node given the name of a field, the path evaluator invokes *JsonDomGetFieldValue()*. The OSON DOM implements this by performing a binary search for the field name identifier over the child node array and then returning the corresponding child node address. For wildcard field steps, it simply returns the array containing all the child node addresses.
For array index path steps, the path evaluator invokes *JsonDomGetArrayElement()*. The OSON DOM implements this by directly accessing the desired position in the array and returning the child node.
In either case, the path engine then evaluates any predicate expression for the current step using each returned child node address as the context node. If the predicate evaluates to true, the path engine can recursively evaluate the next path step by again using the child node address as the context node.
The path engine invokes *JsonDomGetScalarInfo()* for each path predicate expression that reads a scalar value and applies built-in functions or range comparison operators as necessary. The OSON DOM implements this by returning a pointer within the leaf-scalar-value segment, computed using the offset from the node table.
The JSON_TABLE virtual function is designed as a built-in SQL iterator that supports the row source API [9], with implementations of start(), fetchNextBatch(), and close(). For each input document, a set of relational rows are computed by the JSON table driver and returned in response to fetchNextBatch() calls. The JSON table driver calls the DOM based path engine on the SQL/JSON path expression used for row generation to yield a set of tree node addresses. For each node, the path engine is called to evaluate the column path expressions to compute column value. If a NESTED PATH construct is used, the JSON table driver is called recursively on each nested row path expression.

5.2 SQL/JSON query evaluation leveraging Oracle Database in-memory Store Architecture

5.2.1 In-Memory Virtual column for JSON Scalar Value: Oracle 12cR1 supports IMC to improve OLAP query performance by loading rows in memory and organizing the information into an optimized columnar format suitable for SIMD processing [19]. In 12cR2, IMC has been extended to support virtual column expression as well. The JSON DataGuide can help users to discover and define virtual columns using JSON_VALUE() to extract singleton scalar values from JSON documents as discussed in section 3.3.1. Then just as normal SQL columns, JSON virtual columns can map directly to the in memory columnar format for fast SQL in-memory predicate, aggregation, group by and projection operations.

5.2.2 In-Memory OSON for any SQL/JSON query: For columns which have IS JSON check constraints, we automatically add a hidden virtual column defined using the OSON() constructor. Because it’s a virtual column, the OSON bytes aren’t stored on disk. However, when such a table is loaded in memory, the OSON() constructor is implicitly invoked to encode the textual JSON document column into OSON bytes which are then loaded in memory. During query compile time, SQL/JSON queries over the JSON textual column are transparently rewritten to access the OSON virtual column instead. During query execution, if an OSON binary is available in memory, it’s used in lieu of the original textual JSON to evaluate the query more efficiently.

6. Performance Evaluation

We perform the performance evaluation with following criteria.

- For JSON textual documents, we remove all the non-significant white spaces so as to get the smallest possible JSON representation; this minimizes I/O and memory overhead and reduces the JSON text parsing overhead.
- We don't create any indexes over JSON document collections, and we configure enough memory so that the collections are cached entirely in memory. This ensures that any differences in SQL/JSON query processing efficiency among the different encoding methods are attributable solely to differences in their encoding formats.
- For all figures that show performance numbers, the y-axis shows the execution time in a time unit. Please keep in mind that the purpose of these experiments is not to demonstrate the absolute performance numbers that can be achieved, but rather comparing the **performance ratio** among different approaches to evaluate the relative performance characteristics of each approach.

Other than the NOBENCH[6] benchmark discussed in section 6.4, all other performance experiments are conducted on PC running Linux kernel 2.6.18 with a 2.53 GHz Intel Xeon CPU and 6GB of main memory.

6.1 JSON, BSON, OSON Size Statistics

We use various JSON collections including customer use cases and benchmark data from NOBENCH[6] and YCSB[31]. In our experience, customer data sets consist mostly of small to medium size JSON documents with a few large size JSON documents. The medium size JSON document (5MB) is a message archive for twitter postings. The large size JSON document (41.5MB) is from sensor data recording. Table 10 shows the average byte size per JSON document encoded using JSON, BSON and OSON formats respectively. JSON text is UTF-8 encoded with non-significant white space removed. All JSON documents are mainly ASCII characters with 1 byte per character encoding so that JSON text document is measured with its most optimal size. *For small documents, the JSON, BSON and OSON representations are of similar size. For medium and large size documents, the BSON and OSON representations are noticeably smaller than JSON. OSON is much smaller, especially for large documents.* This is because large documents typically have repetitive sub-structures in embedded arrays., The OSON encoding method has an advantage over the other formats because it stores repeated field names just once, in its field-id-name-dictionary segment.

Table 11 shows the average percentage of total space used by the three segments of the OSON format. For small documents, the field-id-name-dictionary segment typically occupies 40% of the total encoded output. For medium to large JSON documents with many structural repetitions, the relative size of field-id-name-dictionary segment is a small fraction of the total. We designed OSON to be self-contained in the same way as JSON text or BSON binary because this avoids many scalability and administrative problems that arise when a central mapping has to be maintained. In our experience, managing, synchronizing, merging, and evolving such a central dictionary is not scalable in an agile distributed computing environment.

JSON Document collection	**Average JSON Text Size per document in bytes**	**Average BSON binary Size per document in bytes**	**Average OSON binary Size per document in bytes**
workOrder	933	1000	952
salesOrder	670	710	732
eventMessage	1924	2117	2113
purchaseOrder	1117	1214	1080
bookOrder	2107	2283	1863
LoanNotes	5146	5353	5710
Twitter Msg	2974	2824	2951
AcquisionDoc	5904	6360	5462
NOBENCHDoc	533	551	654
YCSBDoc	1145	1160	1213
TwitterMsgArchive	5.05M	3.3M	2.5M
SensorData	41.5M	37.4M	18.9M

Table 10 – Avg Size with JSON, BSON, OSON encoding

JSON Document collection	**Average Field-id-name-dict Seg Ratio**	**Average Tree-Node Navigation Seg Ratio**	**Average Leaf-scalar-value Seg Ratio**
workOrder	34.55%	29.92%	35.52%
salesOrder	47.74%	25.94%	26.32%
eventMessage	41%	26.12%	32.65%
purchaseOrder	31.88%	26.89%	41.23%
bookOrder	43.09%	36.99%	19.91%
LoanNotes	62.68%	14.58%	12.29%
Twitter Msg	43.68%	17.61%	38.7%
AcquisionDoc	19.41%	23.51%	57.07%
NOBENCHDoc	40.56%	21.01%	38.44%
YCSBDoc	9.97%	5.6%	84.43%
TwitterMsgArchive	0.05%	42.71%	57.23%
SensorData	0.01%	80.77%	19.22%

Table 11 – OSON Three-Seg Size Statistics

6.2 JSON DataGuide Statistics

We compute a JSON DataGuide for various JSON collections, generate and query DMDV views starting from the root path '$'. Table 12 shows the JSON DataGuide statistics. Number of Distinct Paths is the row count of $DG table: it counts all the distinct JSON paths from the root to both intermediate and leaf nodes. Number of columns in DMDV counts all the distinct JSON paths from the root to leaf nodes only. It represents how "wide" the full master-detail expansion of the JSON hierarchy of the document collection is. DMDV-fan-out ratio, which is computed as the number of rows in DMDV divided by the number of documents in the JSON collection, represents how "tall" the full master-detail expansion of the JSON hierarchy of the document collection is. For medium to large JSON documents, the large DMDV-fan-out ratio occurs because the documents contain large arrays of repeated structures. Also note for NOBENCH [6] data, there are 1000 sparse fields with 11 common fields.

JSON Document collection	**Number of Distinct Paths**	**DMDV – number of columns**	**DMDV-fan-out ratio**
workOrder	29	24	5.5
salesOrder	20	19	3.0
eventMessage	79	55	10
purchaseOrder	29	21	5.0
bookOrder	86	62	11.7
LoanNote	153	122	3.0
TwitterMsg	362	316	1.8
AcquisionDoc	88	73	28
NOBENCHDoc	1011	1000	2
YCSBDoc	10	10	1
TwitterMsgArchive	316	267	5405
SensorData	68	63	32100

Table 12 – JSON DataGuide Statistics

6.3 Performance Comparison of OLAP query between JSON, BSON, OSON and relational storage methods

We used the purchaseOrder JSON collection. Table 1 shows a simplified sample of documents from this collection. We stored

100,000 documents using four storage methods. Figure 4 shows the storage size of each of these methods.

- JSON storage: storing JSON text in a varchar2(4000) JSON column with total storage of 136MB
- BSON storage: storing BSON binaries in a raw(4000) JSON column with total storage of 160MB
- OSON storage: storing OSON binaries in a raw(4000) JSON column with total storage of 136MB.
- REL storage: decomposing each purchaseOrder document relationally and storing the data in two relational tables: *purchase_master_tab* which stores all the top-level purchaseOrder scalar fields and *lineitem_detail_tab* which stores all the lineitems fields. The *lineitem_detail_tab* is linked with *purchase_master_tab* by a foreign key. Total storage of both tables plus primary key and foreign key indices is 112MB.

Table 13 shows the 9 OLAP queries for performance evaluation. These 9 queries are from customer OLAP application queries referencing two relational views *po_mv* and *po_item_dmdv* that serves as an abstraction to hide the underlying physical data storage models differences: schema-rigid relational storage (REL) and schema-less self-contained storage (JSON,BSON,OSON). *po_mv* view projects all the singleton scalar fields from the purchaseOrder whereas *po_item_dmdv* projects all fields from the purchaseOrder with master fields repeated for each detail field. For REL storage, *po_item_dmdv* is defined as a join of *purchase_master_tab* and *lineitem_detail_tab* representing de-normalized master-detail records. For JSON, BSON, and OSON storages, both *po_mv* and *po_item_dmdv* views are defined using JSON_TABLE() over the base JSON column. *Po_item_dmdv* requires one NESTED PATH to un-nest the item array in the same way as the DMDV view definition shown in Table 8. The views are dynamically computed during query processing time by invoking the JSON text parser, BSON decoder, or OSON decoder respectively. The WHERE predicates on the views are pushed down as JSON_EXISTS() with JSON path predicates to be filtered.

Q1, Q2 query only *po_mv*. Q3-Q9 queries query *po_item_dmdv*. Figure 3 Y-axis shows the query execution time for each query for each storage method. Query performance over OSON binary storage is faster than BSON storage which in turn is faster than JSON storage. BSON is marginally faster than JSON. For Q2, Q3, Q4, Q5, Q6, OSON storage is 5 to 10 times faster than JSON storage. Furthermore, OSON storage is on-par with REL storage for all queries. For querying *po_mv*, OSON storage is similar to REL storage to extract the top-level scalar fields. For querying *po_item_dmdv*, OSON storage avoids the need for a join because the master/detail information is de-normalized, whereas REL storage needs to use a hash join to join the master and detail table. Figure 4 shows the storage size comparison among JSON, BSON, OSON and REL. BSON is marginally bigger than JSON and OSON. JSON and OSON are of similar size, both of which are about 21% bigger than REL storage. This is expected because REL stores only data in the table rows, while storing schema in a central relational dictionary, while all other formats are obliged to store schema with each document. The 21% storage overhead is a reasonable trade-off for having self-contained schema flexibility store in exchange for centralized schema management system.

This performance experiment demonstrates that the OSON binary encoding format for JSON bridges the performance gap between schema-rigid relational row storage format and schema-less self-contained text storage format. OSON is self-contained but is more rapidly navigated than textual JSON. Its size is on par with JSON text for small JSON documents and is typically much smaller than JSON text for large JSON text documents because such documents commonly have repeated structures. *Therefore, OSON is an ideal JSON document binary instance format to bridge the performance gap between schema-based encoding and schema-free encoding.*

Q1	select count(*) from po_mv p where p.reference = ?
Q2	select costcenter, count(*) from po_mv group by costcenter order by 1
Q3	select costcenter, count(*) from po_item_dmdv where PARTNO = '97361551647' group by costcenter
Q4	select REFERENCE, INSTRUCTIONS,ITEMNO, PARTNO, DESCRIPTION, QUANTITY,UNITPRICE from po_item_dmdv d where REQUESTOR = ? and d.QUANTITY > ? And D.Unitprice > ?
Q5	select l.Reference, L.Itemno, L.Partno, L.Description from DMDV l where l.PARTNO in (?, ?, ?)
Q6	select Partno, Reference, Quantity, QUANTITY - LAG(QUANTITY,1,QUANTITY) over (ORDER BY SUBSTR(REFERENCE,INSTR(REFERENCE,'-') + 1)) as DIFFERENCE from po_item_dmdv where Partno = ? order by SUBSTR(REFERENCE,INSTR(REFERENCE,'-') + 1) desc
Q7	select sum (quanTity * unitprice) from po_item_dmdv group by costcenter order by 1
Q8	select REFERENCE, INSTRUCTIONS,ITEMNO, PARTNO, DESCRIPTION, QUANTITY,UNITPRICE from po_item_dmdv where quantity > ? and unitprice > ?
Q9	select REFERENCE, INSTRUCTIONS,ITEMNO, PARTNO, DESCRIPTION, QUANTITY,UNITPRICE from po_item_dmdv

Table 13 – OLAP queries

6.4 Performance Evaluation of integration of JSON with in-memory Store and Query Processing

We perform experiments to verify the orders of magnitude performance improvement by integrating JSON/OSON processing with Oracle 12.2 in-memory store. We show that it is feasible to store JSON text on disk but load its equivalent OSON binary into in-memory store so that all SQL/JSON queries can be evaluated using OSON transparently. Furthermore, singleton scalar fields of JSON document can be projected as columnar format and loaded into in-memory columnar store (IMC) for genuine columnar processing. We use NOBENCH benchmark [6] data and run all queries Q1 to Q11. NOBENCH represents a genuine semi-structured document collection with several common fields and many sparse fields. The collection has more than 1000 sparse fields so that it would exceed 1000 column limit imposed by Oracle relational storage if relational de-composition of NOBENCH data were attempted.

The experiment uses 64 million NOBENCH JSON text documents stored in varchar(4000) column and runs 11 NOBENCH SQL/JSON queries. The experiment is conducted on system using Intel Xeon with 16 CPU cores and 256GB DRAM running Linux and Oracle 12.2 Server. The experiment runs in 3 modes.

TEXT-MODE: All JSON text documents are fully cached in Oracle buffer cache and then run 11 queries over JSON text stashed in buffer cache.

OSON-IMC-MODE: OSON bytes for all JSON text documents are fully populated in IMC cache and then run 11 queries over OSON bytes stashed in Oracle IMEC.

VC-IMC-MODE: Following 3 virtual columns (VC) are added into the table:

JSON_VALUE(jobj,'$.str1')
JSON_VALUE(jobj, '$.num' RETURNING NUMBER)
JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER)

and load VC into IMC cache so that Q6,Q7,Q10,Q11 which use the above JSON_VALUE() functions can be processed using columnar format of scalar result from JSON_VALUE().

Figure 5 shows that OSON-IMEC-MODE has significantly improved performance compared with TEXT-MODE. This demonstrates that OSON can be used as an efficient in-memory JSON binary format to transparently improve ad-hoc SQL/JSON query performance despite the JSON is stored on disk in textual form. The advantage of using OSON as in memory format instead of being a persistent on disk format is that we can continuously improve binary encoding format for JSON without maintaining backward compatibility with on-disk format. Similar to Oracle IMC approach that avoids introducing new columnar storage, we've learned a key insight, that is, *decouple the storage format from in-memory query friendly format.* This insight is a principle to support trend of No-DB style In Situ query processing [23] where storage format is uncontrollable by database but any in-memory just-in-time data structure can be constructed and cached for query processing.

Figure 6 shows that VC-IMEC-MODE has significantly improved performance compared with OSON-IMEC-MODE for Q6, Q7, Q10, Q11 where the JSON_VALUE() on three singleton fields: '$.str1', '$.num', '$.dyn1' used in these queries have already been cached in IMC cache so that predicates and projections on JSON_VALUE() can leverage full performance power of Oracle IMC architecture [19].

6.5 Performance Evaluation of document insertion into JSON collection with enabled DataGuide maintenance

We use NOBENCH [6] data to measure the performance impact of maintaining the DataGuide over newly inserted JSON documents. As discussed in section 3.2, we have integrated the DataGuide maintenance with the *IS JSON* check constraint so that inserting JSON documents has minimal overhead in the common situation where no schema changes are discovered. We performed an experiment where we insert 10,000 NOBENCH JSON documents with identical structures. We repeated the experiment three times, each time in a different operating mode.

In "no-json-constraint" mode, we do insertion without the IS JSON check constraint; this measures the base cost of row insertion.

In "json-constraint" mode, we do the same insertion on the table with IS JSON check constraint enforced. This measures the base cost of insertion with the additional cost of reading and parsing the JSON data.

In "json-constraint-dataguide" mode, we turn on JSON DataGuide maintenance so that in addition to reading and parsing the JSON data, we also perform JSON path structural analysis to see if any schema changes need to be written to the $DG table of the JSON search index. Since the documents are identical, no writes to $DG are necessary after the first document is inserted, so this measures the cost of validating the dataguide.

Figure 7 shows insertion time for the 3 modes. With IS JSON check constraint enforced, we measured an overhead of 9.4% compared to not checking the constraint. With JSON DataGuide maintenance added, the overhead climbs to 17%. This means that for homogeneous JSON collections the overhead of the DataGuide is 7.6% versus simply enforcing the IS JSON constraint.

Our second experiment is to compare dataguide maintenance overhead for a heterogeneous JSON document collection with a homogeneous JSON document collection. In the experiment labelled homo, we inserted 10,000 JSON documents with identical structures. In the experiment labelled hetero, we inserted 10,000 JSON documents with every document adding a unique new field so that every insertion of a JSON document will cause a new JSON path to be inserted into the $DG table. Figure 8 shows that the heterogeneous collection incurred doubled the insertion cost of the homogeneous collection.

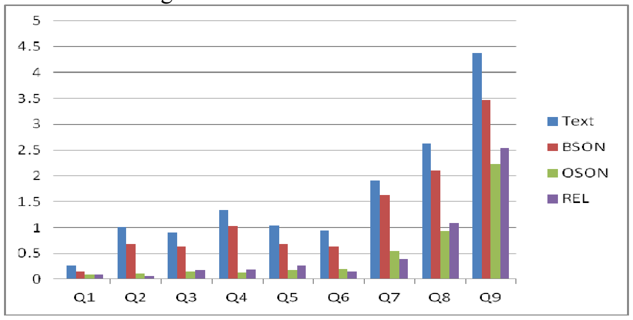

Figure 3 – Query Time Comparison: JSON, BSON, OSON, REL storages

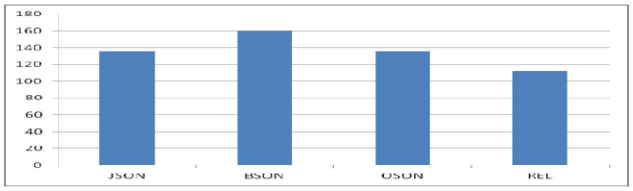

Figure 4 – Storage Size Comparison: JSON, BSON, OSON, REL

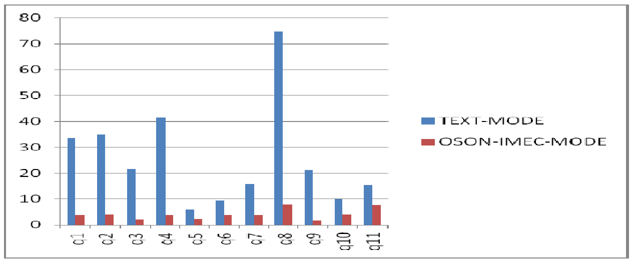

Figure 5 – NOBENCH 11Query Time between TEXT-MODE and OSON-IMC-MODE

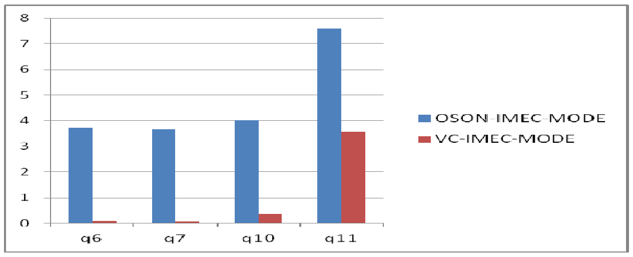

Figure 6 – Q6,Q7,Q10,Q11 11Query Time between OSON-IMEC-MODE and VC-IMC-MODE

6.6 Performance Evaluation of Transient DataGuide Aggregation

We loaded 2 million NOBENCH [6] JSON documents in a collection to measure the performance of transient dataguide computation and compare it with the cost of persistent dataguide creation. We ran a JSON_DataGuideAgg() SQL aggregation query with different sample percentage as Q1 in Table 9. Figure 9 shows the query execution time with sampling 25%, 50%, 75%, 99% of documents in the JSON collection. As expected, the execution time is linear with the number of samples used. We then compare the dataguide query execution time from the 99% sample with the creation time for the JSON search index, which computes a persistent dataguide over the collection. The persistent dataguide takes 27% more time than the transient dataguide. This is expected because both of them performed the same dataguide computation processing over almost the same number of samples, but the persistent dataguide did the additional work of persisting the information to the $DG table.

7. Future Work

Current OSON format is self-contained and instance based resembling a row format without relying on central schema registration and maintenance. We will work on OSON set encoding resembling a columnar format, however, *without reply on central schema definition and assuming homogenous instance collection*. This is different from the design of Dremel [24] which is a columnar encoding of hierarchically nested data conforming to a fixed homogenous hierarchical schema. Although Dremel [24] schema allows optional fields in the schema, it assumes an object field of all object instances within a collection having a fixed position and having the same datatype. For example, it cannot support the case that in one object instance, field 'name' is a string, in second object instance, field 'name' is an integer, in third object instance, field 'name' is a nested object, in fourth object instance, field 'name' is an array. However, such heterogeneous object instance collection is allowed in a JSON collection. Therefore, we want our OSON set format to be able to handle both homogeneous and heterogeneous JSON collections for the in-memory columnar store. Set OSON encoding may explore homogeneity for collection encoding if it can. For example, the common field-id-name dictionary segments can be extracted from each OSON instance and merged into a single dictionary in the in-memory store. This would reduce memory consumption and improve query performance because field name to id mapping can be done once for the entire in-memory store.

Furthermore, the DMDV generated by JSON DataGuide is structurally similar to the wide table [30]. Having many NULL columns and repeated master records per detail record, DMDV is ideal for in-memory columnar dictionary encoding and query processing. We will explore the design of encoding JSON collection as columnar DMDV format and loaded into in-memory columnar store.

In summary, our vision is: On-disk OSON is instance encoded and is self-contained and immune from schema birth and evolution problem to support 'Schema-less for write' No-SQL requirement. In-memory OSON is set encoded based on JSON Data-Guide and is non-self-contained to support 'Schema-Rich for declarative query' SQL requirement.

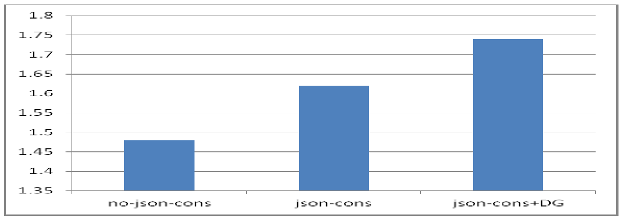

Figure 7 – insertion Time among no-json-chcck, json-check, json-check with dataguide on

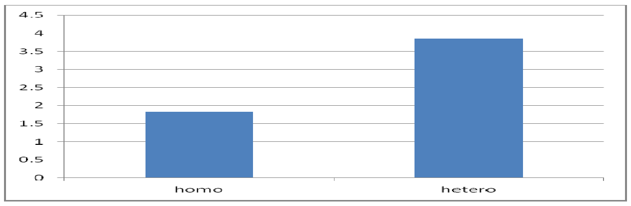

Figure 8 – Insertion Time of Homogenous Versus Heterogeneous Docs with enbled DataGuide

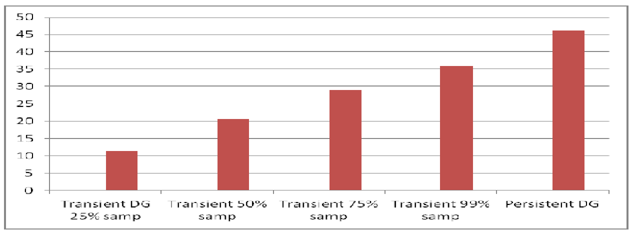

Figure 9 – Query Time for computing Transient JSON DataGuide Aggregation

8. Conclusion

Without JSON DataGuide support, users who adopt a polyglot persistence strategy [7] face two worlds: the relational world with a powerful query language (SQL) but with a requirement to carefully specify and manage schemas, and the NoSQL world with more agile application development and fast data ingestion, but with limited query capabilities. OLTP applications can use a NoSQL database to achieve fast document ingestion. However, afterwards, a relational schema needs to be developed to allow the data to be moved to an RDBMS for SQL analytics. Adding the JSON DataGuide and OSON binary format to the RDBMS shows us that the SQL and NoSQL worlds can be integrated, allowing users to manage both relational data and flexible schema data in a single database engine. The JSON DataGuide offers users a **schema-GPS** to write schema validated query: data schema goes from being hidden inside application data access code to being exposed via the familiar relational model well supported by many tools. OSON format allows JSON to be queried with efficiency that approaches that of schema-based relational tables. Loading OSON implicitly in memory as generic JSON query friendly format have transparently improved query performance.

9. Acknowledgements

We thank Vikas Arora, Asha Makur, Cetin Ozbutun, Mark Drake, George Lumpkin, for SQL/JSON product direction and development management support; Thank Prakashkumar Thiagarajan, Alfonso Colunga, and Rahul Kadwe for helping implementation; Thank Aurosish Mishra, Shasank Chavan, Allison Holloway, Jesse Kamp, Tirthankar Lahiri, Niloy Mukherjee, Vineet Marwah for supporting in-memory virtual column expressions; Thank Brian Chesebro, Srinivas Kareenhalli, Indira Patil, Manoj Sahoo for performance experiment; Thank Dieter Gawlick for evangelizing the concept of flexible schema data. Thank Oliver Kennedy, Boris Glavic for evangelizing future work on derivation of Dataguide from data with uncertainty and provenance tracking.

10. REFERENCES

[1] P. Atzeni, C.S. Jensen, G. Orsi, S. Ram, L. Tanca, R. Torlone: The relational model is dead, SQL is dead, and I don't feel so good myself. SIGMOD Record, 42(2):64-68, 2013

[2] K. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G.M.Lohman, R.Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. C. Truong, B.V. Linden, B. Vickery, C. Zhang: System RX: One Part Relational, One Part XML. SIGMOD Conference 2005: 347-358

[3] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, V. Krishnamurthy: Towards an enterprise XML architecture. SIGMOD Conference 2005: 953-957

[4] R. Cattell: Scalable SQL and NoSQL data stores. SIGMOD Record, 39(4):12-27, 2010.

[5] Chen P.: The Enity-Relationship Model: Toward a Unified View of Data. VLDB 1975: 173

[6] Chasseur, C; Li Y; Patel, J: *Enabling JSON Document Stores in Relational Systems* . WebDB 2013

[7] Fowler, M; Pramod Sadalage, P: *NoSQL databases and Polyglot Persistence:* http://martinfowler.com/articles/nosql-intro

[8] Jagadish, H: *Making Database Systems Usable*. SIGMOD 2007 Keynotes

[9] G. Grafe, Query Evaluation Techniques for Large Databases, in ACM Computing Surveys, 25(2):73–170, 1993.

[10] JSON: http://www.json.org/

[11] Z. H. Liu. "Object-Relational Features in Informix Internet Foundation."Informix technical notes. 9.4(Q4 1999):77-95.

[12] JSONiq: http://www.jsoniq.org/

[13] Stonebraker,M;, Brown,P; Moore, D. *Object-Relational DBMSs*, Second Edition Morgan Kaufmann 1998

[14] Protocol Buffers: http://code.google.com/p/protobuf/

[15] Avro http://avro.apache.org/docs/1.7.5/spec.html

[16] BSON http://bsonspec.org/

[17] Vertica Flex Table https://my.vertica.com/docs/7.0.x/PDF/HP_Vertica_7.0.x_Flex_Tables.pdf

[18] Z.H.Liu, Y. Lu, H.Chang: Efficient Support of XQuery Full Text in SQL/XML Enabled RDBMS. ICDE 2014: 1132-1143

[19] Lahiri, T. et al. Oracle Database In-Memory: A Dual Format In-Memory Database. Proceedings of the 2015 ICDE. Pp. 1253-1258, 2015

[20] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi: Schema Management for Document Stores. PVLDB 8(9): 922-933 (2015)

[21] Z. H. Liu, B. Christoph Hammerschmidt, D. McMahon: JSON data management: supporting schema-less development in RDBMS. SIGMOD Conference 2014: 1247-1258

[22] D. Tahara, T. Diamond, D. J. Abadi: Sinew: a SQL system for multi-structured data. SIGMOD Conference 2014: 815-826

[23] Y. Tian, I. Alagiannis, E. Liarou, A. Ailamaki, P. Michiardi, M. Vukolic: DiNoDB: Efficient Large-Scale Raw Data Analytics. Data4U@VLDB 2014: 1

[24] Dremel: Interactive Analysis of WebScale Datasets S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T.Vassilakis VLDB 2010

[25] http://blogs.msdn.com/b/jocapc/archive/2015/05/16/json-support-in-sql-server-2016.aspx

[26] http://www.info.teradata.com/htmlpubs/DB_TTU_15_00/index.html#page/Teradata_JSON/B035_1150_015K/TeradataSupportJSON.html

[27] https://my.vertica.com/docs/7.0.x/PDF/HP_Vertica_7.0.x_Flex_Tables.pdf

[28] R. Goldman, J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997: 436-445

[29] http://www.ibm.com/developerworks/data/library/techarticle/dm-1501sql-json-db2/index.html

[30] Y. Li, J. Patel:WideTable: An Accelerator for Analytical Data Processing. PVLDB 7(10): 907-918 (2014)

[31] YCSB:https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload

[32] Z.H. Liu, D. Gawlick: Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -. CIDR 2015