JSON Data Management – Supporting Schema-less Development in RDBMS

Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065, USA

{zhen.liu, beda.hammerschmidt, doug.mcmahon}@oracle.com

ABSTRACT

Relational Database Management Systems (RDBMS) have been very successful at managing structured data with well-defined schemas. Despite this, relational systems are generally not the first choice for management of data where schemas are not pre-defined or must be flexible in the face of variations and changes. Instead, No-SQL database systems supporting JSON are often selected to provide persistence to such applications. JSON is a light-weight and flexible semi-structured data format supporting constructs common in most programming languages. In this paper, we analyze the way in which requirements differ between management of relational data and management of JSON data. We present three architectural principles that facilitate a *schema-less development style* within an RDBMS so that RDBMS users can store, query, and index JSON data without requiring schemas. We show how these three principles can be applied to industry-leading RDBMS platforms, such as the Oracle RDBMS Server, with relatively little effort. Consequently, an RDBMS can unify the management of both relational data and JSON data in one platform and use SQL with an embedded JSON path language as a single declarative language to query both relational data and JSON data. This SQL/JSON approach offers significant benefits to application developers as they can use one product to manage both relational data and semi-structured flexible schema data.

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, No-SQL,XML, SQL/XML.

SIGMOD/PODS'14, June 22–27, 2014, Snowbird, Utah, USA.

http://dx.doi.org/10.1145/2588555.2595628

1. INTRODUCTION

RDBMS platforms are based on the relational data model [8] which uses the 'schema first, data later' approach. Before users can load and store any data, they must first design a schema. Not all data fit naturally into the relational model. There are significant amounts of semi-structured data, such as documents, emails, and spreadsheets that do not have a fixed schema or do not have structures that can be effectively modelled as relational schemas. Moreover, skipping the schema design provides productivity gains in today's agile development environments. No SQL systems which support the 'data first, schema later or never' paradigm have rapidly evolved and have a growing customer base.

Semi-structured data management within the RDBMS is available with SQL/XML [5, 14, 23, 25]. However, the XML data format originates from the document processing world and has many rules to represent document-oriented semi-structured data. This complexity makes XML a less than ideal format for representing data-oriented semi-structured data. For data-oriented use cases, JSON [15] has become a popular format because the format is simple and compact and comparatively efficient to process. Native support in languages such as JavaScript has contributed to its popularity. JSON is increasingly the focus of the NoSQL community [3, 6] with products like MongoDB [24] providing native JSON stores.

As JavaScript is heavily used in mobile and web application development (on both clients and servers), the volume of JSON data grows and the trend towards use of NoSQL systems for data persistence accelerates. This trend leads to increased complexity of the management of data with polyglot persistence [11]. Users cannot manage all of their data in one platform; they have to work with different data platforms and deal with the integration of data from these platforms in their application programs. The burden of joining data from different data platforms falls on the shoulders of application developers. Furthermore, developers cannot query all of their data using a single high level declarative query language. Finally, each of the different data management systems has its own operating tasks requiring coordination among administrators.

Our goal is to enable the RDBMS platform to support management of JSON data along with relational data and thereby provide the same advanced and mature data management capabilities to both relational and JSON data. To accomplish this, we need to overcome the challenge that

classical RDBMS platform requires data to have full schema to be storable, queryable, and indexable. This schema-oriented development style needs to be relaxed to support **schema-less development style where data can be stored, queried, and indexed without a schema.** The introduction of SQL/XML for RDBMS took on this challenge but has not been embraced by application developers seeking a schema-less development style. Reasons include the existence of XML schema, the complexity of XML as a data format, and the need to use an alternative query language (XQuery [31]) as an alternative to SQL. SQL/JSON overcomes these problems.

Recall that RDBMS extensibility technologies support user defined types (UDT) [29,21] by storing UDTs in native form without attempting to shred them relationally, by extending SQL with user defined functions (UDF) to query UDTs declaratively, and by indexing UDT with user defined indices to provide efficient access to UDF. Inspired by the extensibility approach, we propose the following three architectural principles for management of JSON data in RDBMS platforms:

- **Storage Principle for JSON**: Leverage the document-object-store model by storing JSON data natively without attempting to shred it into relational form. In this way, JSON data can be stored without schemas and a JSON object collection can contain a variety of JSON object instances without a common schema. Common attributes, if available as a partial schema, can be projected out as a relational view over the JSON object collection.

- **Query Principle for JSON**: Leverage SQL as a declarative **S**et-oriented **Q**uery **L**anguage instead of a specialized **S**tructured **Q**uery **L**anguage just for relational data. Extend SQL with SQL/JSON operators that embed a simple JSON path language to navigate and query JSON object instances.

- **Index Principle for JSON**: Index JSON using partial-schema-aware index methods and schema agnostic index methods. Although functional indexes and materialized relational views can boost query performance based on a partial schema for a JSON object collection, the JSON inverted index can support efficient ad-hoc query access patterns over a JSON object collection without any knowledge of schema information for the collection.

These principles are formulated based on our decades of experience supporting object, XML and full-text indexing using database extensibility technology. Although database extensibility technology [29,21] is well-known as RDBMS engineering practice, we found that abstracting these practices and experiences into the three basic principles helped us to address the challenges of supporting JSON data management to host schema-less development style in RDBMS. To minimize development effort and time to market, it is possible to implement these principles without introducing a new JSON SQL data type. This is in contrast with the SQL/XML standard [14] that requires adding XML as SQL data type in RDBMS. Our SQL/JSON approach is comparatively light weight so that we can support the storage principle, query principle and index principle for JSON data in Oracle RDBMS with minimal effort.

The main **contribution** of this paper is the formulation of the above three JSON data management principles and the design and implementation strategies of these principles in the industry-strength Oracle RDBMS. In addition, the paper also articulates the requirement difference between schema-less JSON data management and schema-oriented relational data management. The requirement clearly justifies why RDBMS needs to apply the three principles to manage JSON data. Supporting SQL/JSON provides a quick yet natural evolutionary path for extending RDBMS to support the schema-less development style that is a de-facto standard in the No-SQL community. Once the RDBMS overcomes the schema-less challenge, the power of SQL as a mature declarative language will be fully unleashed. To the best our knowledge, this is the first industrial paper that articulates the three principles for addressing the challenges of schema-less development in RDBMS via supporting native JSON data management.

Outline: Section 2 compares our work to related work. Section 3 analyzes use case requirement differences between the management of relational data and JSON data to identify what is lacking in the RDBMS platforms to support JSON data management. Sections 4, 5, and 6 present the three principles for JSON support in detail with the design and implementation strategies to support these principles in the Oracle RDBMS. Section 7 presents the results of performance experiments. Section 8 discusses future work. Section 9 concludes the paper with acknowledgements.

2. RELATED WORK

Although XML and JSON are different data formats, at a high level both of them are hierarchical semi-structured data models. Many principles of past work on XML can be applied to JSON. There is a lot of work on XML storage, index and query processing in both academia and industry with major RDBMS vendors supporting the SQL/XML standard in their products during last decade [5, 23, 25]. For XML storage, there are shredded relational mappings with or without the help of XML schema and aggregated native binary XML storage [5, 25, 26]. For indexing XML, there are XPath or XMLTABLE expression based methods which require at least a partial schema of the XML document collection [5, 19]. There is also the path-value based method which avoids any requirement to specify schema information for the XML document collection [23, 33]. For processing of XQuery over XML, there is a query rewrite technique to transform XML query into equivalent SQL with XML extension operators to leverage the underlying physical XML storage and index methods [20].

More recently, there is published work on storing and querying JSON data. Enabling JSON store in Relational System [9] is an approach where JSON objects are decomposed using a schema-less shredding approach into a path-value vertical relational table recording the paths and values of the original JSON object. It proposes Argo/SQL, a SQL like query language for querying JSON objects. Argo/SQL is transformed into equivalent SQL that queries the underlying vertical table. Conceptually, the vertical shredding approach is similar to the path-value based XML Index [23,33] and to the use of a vertical storage table to store flexible schema data [2]. The Argo/SQL to SQL transformation is in principle the same as that of transforming XQuery/XPath into equivalent SQL queries over the path-value XML index table [20]. Paper [9] also defines a NOBENCH benchmark to run a set of queries over a JSON object collection.

The JSON object collection defined by the NOBENCH benchmark models the characteristics of semi-structured data collections very well.

Google F1 database [28] stores protocol buffer format [36], a binary form similar to JSON, and extends SQL with path expressions and a PROTO JOIN construct to query protocol buffers stored in a table. Protocol buffer storage is in principle the same as that of native aggregated binary XML storage without shredding [5, 25]. MongoDB [24] stores JSON in a binary format BSON [38] and provides an API to JSON object collections that can find BSON objects using filter criteria and produce projections of components of BSON objects. MongoDB supports JSON index by projecting out commonly queried fields from the JSON object collection. This is in principle the same as that of XML indexing that requires knowing a partial schema of XML collection [5,19]. Vertica [40] flexible table approach can let user obtain values of common fields from JSON object collection and load them into a relational table. This is basically materializing JSON_TABLE() results declaratively. However, the flexible table does not seem to provide mechanism to load each element of a JSON array into a separate detail table. Like XMLTABLE(), JSON_TABLE() has mechanism to chain the result of array into separate detail table.

Our JSON storage principle is aligned with the approach in MongoDB [24], Google F1 DB [28] and LinkedIn Espresso [39]. We store JSON objects natively, without shredding, avoiding the cost of reconstructing the original JSON object during queries. Our JSON query principle is to extend SQL with SQL/JSON operators that embed a JSON path language in the same way that SQL/XML operators embedded the XQuery language. However, the SQL/JSON path language is a simple path navigation language, not a complex standalone language such as Jaql [4], JSONiq[17] or XQuery[31]. We choose a simple JSON path language because in RDBMS, SQL and PL/SQL (SQL/PSM) have already provided rich set query language constructs and procedural control flow constructs respectively. Our SQL/JSON approach is aligned with the Google F1 DB [28] approach of extending SQL to query JSON. The PROTO JOIN construct in Google F1 DB is essentially equivalent to the XMLTABLE() construct in SQL/XML standard [14] and the JSONTABLE() construct discussed in section 5.2. The JSON object path access is directly inlined in Google F1 DB SQL. Semantically it is a syntactic sugar to SQL/JSON JSON_VALUE() operator to be discussed in section 5.2. Similar to MongoDB [24], SQL/JSON path language has a lax mode to accommodate querying of heterogeneous JSON object collections. This is in contrast with JSONiq [17] that does strict mode only. We will discuss this in section 5.2.

Our JSON index principle supports both schema-aware and schema agnostic indexing. The schema-aware indexing method is conceptually the same as that of functional indexing [29] and XMLTABLE indexing [19] method. The schema agnostic indexing method is functionality the same as XML path-value indexing [23,33] and the Argo-SQL vertical shredding approach [9]. However, we will show that the underlying design of schema agnostic indexing method should be based on inverted index architecture originated in the information retrieval community [27]. LinkedIn Espresso [39] uses Lucene inverted index. However, our approach generalizes the idea of inverted index to not only index keywords but also to index JSON paths and values in JSON object. We will discuss this in section 6.2. Furthermore, unlike index over distributed data service platform [39], our JSON inverted index is a domain index that is consistent with base data just as any other index in RDBMS. Each RDBMS replication site maintains its own index over its data replica.

3. Requirements of JSON data management in RDBMS

There are significant differences between managing relational data and semi-structured data represented as JSON. We analyze these differences from the perspective of JSON data storage, data query, and data index.

3.1 Data Modeling Difference: Schema First Versus Schema Later/Never

RDBMS is originally designed to manage structured data whose structures can be cleanly separated from the content. Structures are defined as schema that is managed as system meta-data by the RDBMS. Entity/Relationship (E/R) modelling [7] is very effective at breaking down structures into tables linked with primary and foreign keys to support typical master-detail join pattern. Structural changes require DDL statements to alter the system meta-data before new data with the changed shape can be loaded. If a schema is not available, data cannot even be loaded. Such 'schema-first, data later' requirements have caused a 'birth-pain' problem in RDBMS [12].

However, not all data fits naturally into the relational model. For a collection of semi-structured data objects, the structure of data is not easily separable from the content because the structure of data varies from instance to instance. A collection of JSON objects may have a small number of common attributes complemented by a large variety of varying attributes resulting in a large number of *sparsely populated columns* [2] when stored relationally. In NOBENCH benchmark [9], *sparse_XXX* is a series of sparsely-populated attributes from *sparse_000* to *sparese_999*. Storing all of them using an E/R model requires a table with over 1000 columns. This may exceed design limits of and RDBMS, and will have large numbers of NULL values on every row. We call this the sparse-attribute issue. A second problem is that an attribute may have a different datatype in different data instances. In NOBENCH benchmark[9], *dyn1* is a dynamically typed value that is either a number or a string. RDBMS does not support multi-typed column natively. We call this the polymorphic typing issue. A third problem is that the cardinality of an attribute may vary from one instance to another, or over time. An attribute may be a singleton value at first and later become a JSON array. For example, a *person* object may initially have one *contact phone number* but is later altered to have an array of *contact phone numbers*. Such singleton to array schema evolution causes data migration issues if we use an E/R design [7] and must storing array data in a separate detail table joined with the master table. We call this the singleton-to-collection issue. Lastly, it is not uncommon for JSON objects to possess recursive structures. This is common in XML constructs as well. Modelling recursive structures in E/R model is feasible but SQL doesn't widely support such constructs as first class operations. We call this the recursive structure issue. In summary, it is often hard to define one relational schema to capture all of the JSON data in a collection

using the classical E/R model - at best, developers may derive some partial schema. In the NOBENCH benchmark [9], the *str1*, *str2*, *num*, *bool*, *nested_obj.str*, *nested_obj.num* can be considered partial schemas as they are common in all JSON object instances and thus their values can be extracted out of JSON objects. Partial schema is not sufficient to decompose all data into relational schema. Instead they can be modelled as virtual columns or relational views on top of JSON object collections. This is the idea of *partial shredding that leverages the relational model as an auxiliary structure on top of a native object collection.*

3.2 Querying Difference: Querying Flattened Data with Static Schema Versus Querying Hierarchical Object with Dynamic Schema

First, using an E/R model, data structures are decomposed relationally by first registering object structural schema as meta-data. Consequently, the structure of the data is known ahead of query execution time so that only data is examined during query execution time. Therefore, SQL does not need constructs that can query the structure of data during query execution. For JSON data whose schema is not registered as system meta-data, the query language needs to have constructs that can query both the structure and data together during query execution. SQL alone is not sufficient - SQL needs to be enhanced to have language constructs that can query both structures and data of JSON objects at run time.

Second, the E-R approach models hierarchies of data using the master-detail pattern, so querying hierarchical data decomposed into a relational model requires the use of explicit joins of primary key and foreign key columns when traversing a hierarchy. However, for JSON objects which are not stored in decomposed manner, it is more natural to express hierarchical traversal using a path navigation language. SQL needs to embrace path navigation constructs for querying JSON objects natively. Although SQL99 supports object type with a path navigation construct, SQL99 still requires the schema of object type to be registered with RDBMS as meta-data. Therefore, SQL99 object type support is still based on a static schema model. JSON object support requires dynamic schemas. Furthermore, when a path language is applied to semi-structured data, it needs to support wildcard steps in the path as well as traversal of hierarchical structures without knowing the exact number levels of descent needed. All of these constructs are supported in XPath 1.0.

Third, while relational queries result in columns values that are always scalar values, path navigation results represent projections of both scalars and sub-structures from underlying JSON objects. SQL needs to support JSON operators that are able to extract or update components of JSON data and are able to construct new JSON data.

Finally, JSON data may contain text content that is useful to index for full text search. However, unlike pure textual documents, the content is embedded inside JSON object members and array elements. JSON full text search needs to incorporate path navigation. Such a keyword search operation is shown as Q8 in NOBENCH benchmark [9].

3.3 Indexing Difference: Partial-Schema-aware Indexing Method Versus Schema agnostic Indexing Method

Index and materialized view are auxiliary structures that are built on top of relational store to speed up queries in RDBMS. However, defining B+ tree index, bitmap index, and materialized views all require relational schema; e.g., their creation is based on the '*schema first, index definition later*' approach.

Partial schema discussed in section 3.1 can be used to define virtual columns and relational views on top of the collection. Virtual columns can be used to construct B+ tree or bitmap indexes, - all of these techniques serve to boost the performance of queries whose patterns are known. Supporting partial schema-aware indexing is a natural consequence of adopting the '*data first, schema later*' approach in RDBMS. However, since a JSON object in a row may have array consisting of multiple values to be indexed, therefore, classical B+ tree index that assumes one indexed value per row is not sufficient to index multiple values within an array of a JSON object. This is known as index cardinality issue.

Ad-hoc queries for exploratory and search purposes must be supported by a JSON indexing method that does not rely on knowing any partial schema for the target collection. Such an indexing method requires both JSON member and array names to be indexed together with the data. Supporting schema-less indexing method is a natural consequence of supporting the '*data first, schema never*' approach in RDBMS.

4. JSON Storage Principle: Support for Document-Object Store Model without Relational Shredding

No shredding of JSON object: To host schema-less development with a collection of heterogeneous JSON objects, we store each JSON object instance as one aggregated object in a column without any relational decomposition. A JSON object collection is modelled as a table having one column storing JSON objects. Each row in the table holds a JSON object instance in the collection. This design embraces the document-object model in No-SQL DB systems [6]. We don't perform the vertical shredding as presented in paper [9]. We will justify this decision from a performance perspective in section 7.

No JSON SQL datatype: We do not introduce JSON as a new distinct SQL datatype. Supporting a new native SQL datatype would require changes throughout the RDBMS kernel and require enhancements to all client APIs, as well as development tools and frameworks. Given widely distributed RDBMS client installations, it would require a client upgrade to use the new JSON functionality. Another issue is that there are already many different JSON-like representations: besides the JSON textual format, there is the BSON binary format [38], the Protocol Buffer binary format [36], and the AVRO binary format [37]. It is not clear which format(s) will be widely adopted. In the future, new formats may be introduced. Finally, JSON data can be stored in file systems external to the RDBMS (such as HDFS) and then mapped as an external table to support SQL queries. In such cases the RDBMS must have the ability to consume JSON data **as is**.

Storing JSON using existing SQL datatypes with Check Constraint: We have chosen to give developers the flexibility of storing JSON data in SQL VARCHAR, CLOB, RAW, or BLOB columns, rather than adding JSON as an abstract SQL datatype as what we have done to support XMLType in the Oracle RDBMS [25]. In lieu of this, we provide an *IS JSON* SQL built-in operator to verify whether the input text or binary is a valid JSON object. Developers can use the *IS JSON* predicate as column check constraint to enforce table column values storing only valid JSON data. T1 of Table 1 shows a DDL statement to define a table storing JSON text capturing shopping cart information with the *IS JSON* check constraint. INS1 and INS2 of Table 1 are two SQL insertion statements to load JSON objects into the JSON object collection table. Oracle VARCHAR and RAW data types can support up to 32K size. Beyond that, the CLOB and BLOB data types can be used to hold large JSON object instances. Oracle LOB column infrastructure can provide compression, de-duplication, and encryption features for LOB data[18] - all of which are available for use with JSON objects to reduce storage size and improve query performance. Small size data improves I/O performance because I/O is still the main performance bottleneck in DBMS [10]. Column level encryption provides additional security access control for JSON objects stored in RDBMS. In short, using existing SQL data types to store JSON ensures full operational completeness for JSON data with minimal effort. This includes features as partitioning, replication, import/export, and bi-temporal features, all of which are critical to support the full data operational life cycle for any data including JSON data. If JSON data were shredded relationally, supporting all of these features on shredded JSON objects would require significant development effort.

Virtual Relational Columns over JSON object collection: For partial schema use cases, data in a JSON object collection can be projected out as virtual columns attaching to the collection table using the JSON_VALUE() operator. See *sessionId* and *creationTime* as virtual columns listed in T1 of Table1. Composite B+ tree indexes can be created on virtual columns to facilitate range queries over projected scalar values.

T1	*CREATE TABLE shoppingCart_tab* *(* *shoppingCart VARCHAR2(4000) check (shoppingCart is JSON),* *sessionId NUMBER AS* *(JSON_VALUE(shoppingCar ,'$.sessionId' RETURNING NUMBER)) VIRTUAL,* *userlogin varchar2(30) AS* *(CAST(JSON_VALUE(shoppingCart,,'$.userLoginId')* *AS varchar2(30))) VIRTUAL* *)*
INS1	*INSERT INTO shoppingCart_tab(shoppingCart) VALUES('* *{"sessionId" : 12345,* *"creationTime": "12-JAN-09 05.23.30.600000 AM",* *"userLoginId" : "johnSmith3@yahoo.com",* *"Items" : [* *{"name": "iPhone5", "price" :99.98, "quantity" :2, "used": true,* *"comment" : "minor screen damage"},* *{"name": "refrigerator", "price" :359.27, "quantity" :1, "weight": 210,* *"Height" : 4.5, "Length" : 3, "manufacter": "Kenmore", "color" : "Gray"}]}* *')*
INS2	*INSERT INTO shoppingCart_tab(shoppingCart) VALUES('* *{"sessionId" : 37891,* *"creationTime": "13-MAR-13 15.33.40.800000 PM",* *"userLoginId" : "lonelystar@gmail.com",* *"Items" :* *{"name": "Machine Learning", "price" :35.24, "quantity" :3, "used" : false,* *"category" :"Math Computer", "weight":"150gram"}}* *')*
IDX	*CREATE INDEX shoppingCart_Idx ON shoppingCart_tab(userlogin, sessionId)*

Table 1 – JSON object collection DDL

In summary, the **storage principle of JSON** avoids shredding objects into relational forms. Instead, each JSON instance is stored in an aggregated form with everything stored together so that it is **self-contained without relying on a central schema dictionary**. This facilitates import/export of JSON data and distributed operation among Oracle RDBMS servers and clients. Aggregated storage also eliminates the cost of reassembling the original data from multiple relational tables via joins.

5. JSON Query Principle: Embedding JSON object query language in SQL

5.1 Leverage SQL as inter-JSON object Set-oriented Query Language to query JSON object collection

Instead of '**S**tructured **Q**uery **L**anguage', we call SQL as '**S**et oriented **Q**uery **L**anguage'. Being a *set-at-a-time* query language, SQL can be used as an *inter-object query language* to query objects stored in the object collection table. Therefore, there is no theoretical reason to construct yet another set-at-a-time query language, such as JSONiq [17], having equivalent set algebra specifically targeted at JSON collections. However, classical SQL assumes the element of a set is one relational row with relational columns where as JSON collection is a set of JSON object instances. To provide declarative navigational access of JSON object instance, we need a JSON path language. The JSON path language is used as an intra-object query language in conjunction with SQL as the overall Set Query Language and Oracle PL/SQL or SQL/PSM as the overall procedural language.

5.2 SQL/JSON Standard

Based on industrial research and development experiences for SQL/XML[14], Zhen Hua Liu originates SQL/JSON standard that defines a set of SQL/JSON query operators that embed a simple JSON path language to provide declarative query language over a collection of JSON objects and a set of SQL/JSON construction functions from pure relational data. The working group of SQL/JSON standard consists of SQL standard representatives and research engineering architects from IBM and Oracle: Beda Hammerschmidt, Krishna Kulkarni, Zhen Hua Liu, Doug McMahon, Jim Melton, Jan-Eike Michels, Hamid Pirahesh, Fatma Ozcan, Fred Zemke. The working group defines a SQL/JSON path language as an intra-object query language over a JSON object instance and exact semantics of SQL/JSON query operators and construction functions.

5.2.1 SQL/JSON query operators

Similar to SQL/XML [14] query operators, SQL/JSON query operators are used in strategic places in SQL as shown in Figure 1. The SQL/JSON operators can query JSON objects stored in VARCHAR, CLOB, RAW, or BLOB columns with proper JSON format clauses. If the input data type is VARCHAR or CLOB, the input is assumed to contain textual JSON. If the input data type is RAW or BLOB, input may contain JSON text in standard utf-8 character encoding, or, using an optional format clause to indicate the input data is in one of the binary formats: BSON, Avro, or Protocol buffer [36,37,38].

SQL WHERE clause for JSON object Search: JSON_EXISTS(), JSON_VALUE() are used in a SQL WHERE clause to search and filter JSON object instances stored in a

JSON object collection. See Table 6 Q3, Q4, Q5, Q6, Q7, Q9, Q10, and Q11 for examples of using these operators. In addition, to support full text search within JSON path navigation as described in section 3.2, **Oracle RDBMS also supports JSON_TEXTCONTAINS(), which is not part of SQL/JSON standard, in the WHERE clause.** See Table 6 Q8 for keyword search within JSON array.

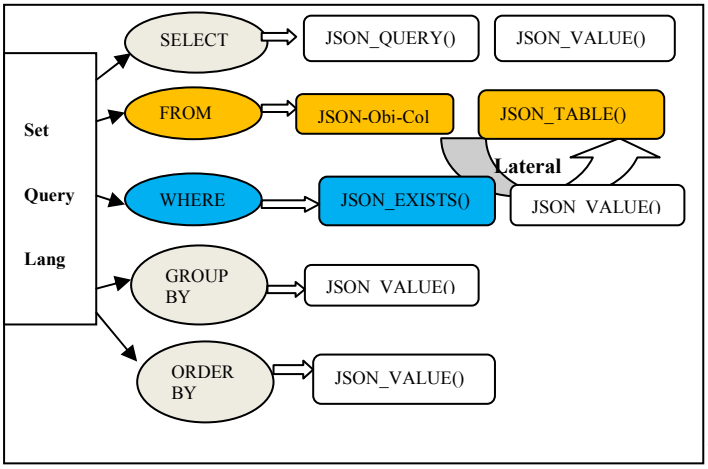

Figure 1 – SQL/JSON operators usage in SQL

SQL FROM Clause to iterate JSON object component: JSON_TABLE() is used in the SQL FROM clause to convert arrays within JSON object instances into a virtual relational table. It is defined as a lateral join with the JSON object collection table. The JSON path language can be used in both row and column expressions of JSON_TABLE(). Table 2 Q2 shows an example. The typical use case of JSON_TABLE() is to expand a JSON array within a JSON object into a set of relational rows, each of which corresponds to an element within the array. JSON_TABLE() acts as a bridge between relational data and JSON data so that islands of structural data within JSON object can be projected out. Effectively, *JSON_TABLE() enables RDBMS users to capture partial schema in a JSON object collection as a set of relational views.*

SQL SELECT, GROUP BY, ORDER BY Clauses to retrieve SQL scalar value from JSON: JSON_VALUE() is used to extract scalar values within the JSON object and cast them into values corresponding to standard SQL built-in types such as VARCHAR, NUMBER, DATE. Extracted values can be used in scalar value expressions in SELECT, GROUP BY, and ORDER BY clauses where scalar values are expected. See Table 6 Q10 GROUP BY clause. In case that extracted scalar value is not castable to the desired datatype, error handling options, such as NULL ON ERROR, DEFAULT value ON ERROR, ERROR ON ERROR, are provided. By default, it is NULL ON ERROR option which makes JSON_VALUE() to return SQL NULL value in case of error so as to gracefully handle polymorphic typing issue discussed in section 3.1.

SQL SELECT Clauses to project JSON object component from JSON object: JSON_QUERY() is used to extract complex sub-components from within JSON objects or to construct new JSON objects so that they can be used in a SELECT clause. See Table 2 Q1. Unlike JSON_VALUE(), whose return value is always a SQL scalar, JSON_QUERY() returns a JSON object or JSON array. However, since we don't introduce a SQL JSON data type, JSON_QUERY() supports a return clause to specify the SQL data type, such as VARCHAR or LOB to hold the JSON_QUERY() result.

SQL UPDATE clause of JSON Object Collection: JSON object collections can be updated using regular SQL Delete, Insert and Update statements. JSON_EXISTS() is typically used in the WHERE clause to qualify documents or objects to be deleted or updated. The right hand side of assignment in UPDATE clauses can be any SQL expression that constructs a new JSON object instance to replace an existing JSON object. See Table 2 Q3 for an example of an update. Future work in SQL/JSON standard will allow JSON_QUERY() used as the right side expression of a SQL UPDATE statement to replace an existing JSON object with a new object by applying updating transformation expressions on the existing JSON object. This is conceptually the same as the XQuery Update Facility in SQL/XML that can update XML documents stored in an XML document collection [22]. All of these enhancements will be part of the process for the evolving SQL/JSON standard.

SQL JOIN among Object Collections: In No-SQL system, join operations are typically not supported. Because the document storage model does NOT shred and normalize objects into nested tables linked by primary and foreign key, intra-object joins are not needed. However, there is still need to do inter-object join across multiple JSON object collections. No-SQL system seems to lack of such join capabilities and require application programmers to perform any needed joins in their application code. In SQL, joins can be performed directly, leveraging the full power of the RDBMS to optimize the query, and ensuring that unselective parts of joins are not needlessly transported to the client application. See Table 6 Q11 for an example of a self-join on a JSON object collection. See Table 2 Q4 for an example of a join across different JSON object collections. In this case, *customerTab* is a table with a *customer* column storing JSON object instances representing customers. Moreover, joins are not restricted to JSON object collections - users can freely join against XML document collections or traditional relational tables.

5.2.2 SQL/JSON Path Language

The SQL/JSON path language is structurally similar to XPath 1.0 and JsonPath [16]. Currently, SQL/JSON path language has only path step expressions with filter expressions used as predicates for path steps. Each path step expression is either the JSON object member accessor or the JSON array element accessor. In the near future, SQL/JSON standard may extend the language to have constructor expressions and update expressions to do component-wise updates of JSON objects. Here, we present some subtle issues in designing the SQL/JSON path language.

Q1	*SELECT p.sessionId, **JSON_QUERY**(p.shoppingCar, '$.items[1]' RETURN AS VARCHAR(2000))* *FROM shoppingCart_tab p* *WHERE **JSON_EXISTS**(p.shoppingCar , '$.item?(name="iPhone"))')* *ORDER BY p.userlogin*
Q2	*SELECT p.sessionId, p.userlogin, v.Name, v.price, v.Quantity* *FROM shoppingCart_tab p,* ***JSON_TABLE**(p.shoppingCart , '$.items[*]'* *COLUMNS* *Name VARCHAR(20) PATH '$.name',* *price number PATH '$.price',* *Quantity integer PATH '$.Quantity') v*

Q3	***UPDATE*** *shoppingCart_tab p* *SET p.shoppingCart_tab = <SQL-Expr constructing JSON>* *WHERE* ***JSON_EXISTS****(p.shoppingCart , '$.item?(name="iPhone"))')*
Q4	*SELECT COUNT(*) FROM customerTab p, shoppingCart_Tab p2 WHERE JSON_VALUE(p.customer, '$p.contact-info.email-address') = JSON_VALUE(p2.shoppingCart, '$."userLoginId"')*

Table 2 – SQL/JSON Query Example

Sequence Data Model: XML documents are not a complete data model for query processing. Likewise, JSON objects and arrays are also incomplete as a foundation for processing. A path language requires expression results to form a closure under the chosen data model. Therefore, the SQL/JSON path language defines a JSON data model which is a sequence of items, each of which is a JSON data model item. Each item can be either a JSON object, JSON array, JSON atomic value. The JSON atomic value can be of string, number, or boolean type. In addition, the atomic value can be of date, time, timestamp, or interval values using the semantics of the equivalent SQL built-in types. Note that JSON array is not used to represent the sequence data model because a JSON array may be an item in the sequence data model, and arrays can be nested whereas sequences cannot. JSON sequence data model follows the same algebraic property as that of the sequence data model in XQuery [31] and JSONiq [17]. That is, it does not support nested sequences. A singleton item is equivalent to a sequence of that singleton item.

Predicate Filter expression: Unlike XQuery that introduces cumbersome rules on effective boolean value for any sequence, SQL/JSON path language differentiates filter expressions from path expression so that filter expressions can be used only as predicates of path expressions. *'$.items?(exists(weight) && exists(height))'* is the SQL/JSON path expression to test if an *item* has both *weight* and *height* member. The exists() function is explicitly used to convert a set into a boolean value by testing the emptiness of the set. This design is similar to that of SQL which uses EXISTS() subquery to test the emptiness of a set.

Lax Mode for object and array accessor: An important difference between XML and JSON is the treatment of collection-valued items. In XML, there is no special construct to differentiate a singleton element node and a collection of element nodes having the same name - elements can simply be repeated. Therefore, to query either a single element node with element name '*item*' or a collection of element nodes with the same element name '*item*', the same XPath step '/*item*' is sufficient. In contrast, JSON has an explicit array construct to hold multi-valued properties. A singleton item is therefore distinguished from an array with a single value. Because of this, querying singleton items and array items from a JSON collection require different path expressions. Consider the two JSON objects inserted as INS1 and INS2 of Table1. To access the first item in the *shoppingCart* of JSON object in Tabel1.INS1, the JSON path language expression is *'$.items[1]*' because *items* is an array. To access the first item in the *shoppingCart* for JSON object in Tabel1.INS2, the JSON path language expression is *'$.items*' because items is not an array. This makes writing JSON path language to handle singleton-to-collection issue discussed in section 3.1 for JSON object collection quite complicated. It also poses a challenge for schema-less development in that the author of a query may not know which construct to use. To solve this, the SQL/JSON path language uses lax mode. In lax mode, there is an implicit wrapping and unwrapping operation applied at each step in the path. When an array element accessor is applied to a JSON object item, a singleton item is implicitly wrapped as an array of that item. When an object member accessor is applied on a JSON array item, the array is implicitly unwrapped so that the object member accessor is applied to each item of the JSON array. This is in contrast with JSONiq [17] which does not provide such lax mode flexibility.

Lax Error Handling: The SQL/JSON path language is more forgiving of errors by returning false in filter expressions instead of raising error. Such forgiving behaviour is important to handle the polymorphic typing issue discussed in section 3.1. Consider applying the JSON path expression *'$.items?(weight > 200)*' to the JSON object in INS2 of Table 1 - the value of "*weight*" is '*150gram*' which is not a number comparable with 200. Rather than raising a type error, this results in false. Such lax mode of error handling is in contrast with JSONiq [17] that does not provide such flexibility for error handling.

5.3 Design and Implementation Strategies of SQL/JSON in Oracle RDBMS

To ensure that the SQL/JSON path language can be evaluated efficiently, we support streaming processing without materializing entire JSON objects in memory. Figure 4 shows the JSON event stream flow diagram for SQL/JSON Path language processor.

Our JSON text parser and JSON binary decoders generate a JSON event stream. The JSON event stream is conceptually similar to that of an XML SAX event stream. It is composed of BEGIN-OBJ, END-OBJ, BEGIN-ARRAY, END-ARRAY, BEGIN-PAIR, END-PAIR, and ITEM events. The BEGIN-PAIR and END-PAIR event wraps a JSON member name and its content pair. The name of the member can be obtained from the BEGIN-PAIR event. The ITEM event represents a typed scalar value that is within the BEGIN-PAIR and END-PAIR event, or within an array. Each JSON path expression is compiled into a state machine to listen and process the JSON event stream. The SQL/JSON path language processor can process multiple JSON path expressions as the same JSON event stream from the source JSON object can be consumed by multiple JSON state machines. This situation is typical of the multiple path expressions found in the JSON_TABLE() operator. For efficiency, we implemented JSON_VALUE(), JSON_EXISTS(), JSON_QUERY() as RDBMS server built-in kernel operators, rather than as user defined functions. JSON_TABLE() is built as a RDBMS server kernel row source that is processed iteratively and corresponding to the overall SQL iterator row source design [13]. The streaming evaluation model of SQL/JSON path language processor fits with the SQL iterator row source design very well. We use a lazy evaluation strategy for SQL/JSON operators by pulling items from the JSON path processor in a streaming fashion. For JSON_EXISTS(), as soon as there is one item returned from the JSON path processor, it can return TRUE and the evaluation stops. For JSON_TABLE(), it pulls the item from the JSON path processor as the parent row source demands it. For JSON_VALUE(), it is an error if there is more than one item. Only JSON_QUERY() needs to aggregate items flowing from the JSON path processor .

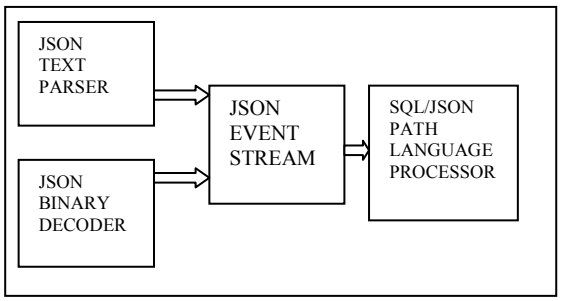

Figure 4 – SQL/JSON PATH PROCESSOR

Both the row and column JSON path expressions of JSON_TABLE() are all passed to the JSON path processor so that all of them can be evaluated simultaneously from a common JSON event stream. Furthermore, if JSON_TABLE() is not outer joined with the JSON object collection, then no row is returned for the JSON object in the JSON object collection if the row expression returns no item. This is equivalent to attaching a JSON_EXISTS() operator using the row expression of JSON_TABLE() to the WHERE clause of the JSON object collection. This can improve performance significantly if an index can be used to evaluate the JSON_EXISTS() operator. See T1 transformation rule in Table 3. Multiple JSON_VALUE() operators applied on the same JSON object input can be transformed into one JSON_TABLE() construct during compile time. See transformation rule T2 in Table 3. Multiple JSON_EXISTS() operators on the same JSON object input column can be combined into one JSON_EXISTS() operator. See transformation rule T3 in Table 3. Both T2 and T3 transformation is done so that we can share the evaluations of multiple JSON path expressions by streaming the JSON object once during run time.

Qry#	Original Query	Transformed Query for optimization
T1	*SELECT p.sessionId, p.creationTime, v.Name, v.price, v.Quantity FROM shoppingCart_tab p,* ***JSON_TABLE****(p.shoppingCart , '$.items[*]' COLUMNS Name VARCHAR(20) PATH '$.name', price number PATH '$.price', Quantity integer PATH '$.Quantity') v*	*SELECT p.sessionId, p.creationTime, v.Name, v.price, v.Quantity FROM shoppingCart_tab p,* ***JSON_TABLE****(p.shoppingCart , '$.items[*]' COLUMNS Name VARCHAR(20) PATH '$.name', price number PATH '$.price', Quantity integer PATH '$.Quantity') v* ***WHERE JSON_EXISTS(p.shoppingCart, '$.items[*]')***
T2	*SELECT JSON_VALUE(p.shoppingCart , '$.items[1].name'), JSON_VALUE(p.shoppingCart , '$.items[1].price RETURNING NUMBER), JSON_VALUE(p.shoppingCart , '$.items[2].name), JSON_VALUE(p.shoppingCart , '$.items[2].price RETURNING NUMBER), FROM shoppingCart_tab p*	*SELECT v.item1_nm, v.item1_price, v.item2_nm, v2.item2_price FROM shoppingCart_tab p,* ***JSON_TABLE****(p.shoppingCart , '$.items' COLUMNS vitem1_nm VARCHAR(20) PATH '$[1].name', vitem1_price NUMBER PATH '$[1].price', vitem2_nm VARCHAR(20) PATH '$[2].name', vitem2_price NUMBER PATH '$[2].price')*
T3	*SELECT count(*) FROM shoppingCart_tab p WHERE JSON_EXISTS(p.shoppingCart , '$.item?(name="iPhone"))) AND JSON_EXISTS(p.shoppingCart , '$.item?(price > 100)')*	*SELECT count(*) FROM shoppingCart_tab p WHERE* ***JSON_EXISTS****(p.shoppingCart,* ***'$?(item? (name="iPhone") and item?(price > 100))')***

Table 3 – SQL/JSON Transformation Rule for Optimization

6. JSON Index Principle: Partial Schema-Aware Index Method & Schema-Argnostic Index Method

6.1 Partial Schema-Aware Index Method

Situations where common path expressions or members can be identified for a collection are referred to as partial schemas. Such partial schemas can be projected out as auxiliary structures on top of the original JSON object collection in the form of functional indexes, virtual columns to speed query performance. The simplest indexing method is functional indexing using JSON_VALUE() to facilitate range searches on the result of JSON_VALUE(). If multiple members from a JSON object collection need to be range queried together, we can create a composite B+ tree index over multiple virtual columns, each of which is a projection of a member. IDX of Table 1 illustrates this approach.

Recall index cardinality issue in section 3.3, JSON objects commonly include arrays representing master-detail relationships. For example, a *shoppingCart* item stores a collection of items. To index all values in a JSON array, we use JSON_TABLE() that projects out master-detail relationship as shown in Table 2 Q2. To avoid storing repeated master records for each detailed item, we support a construct similar to the widely used XML Table index [19]. The table index internally creates master-detail relational tables to hold the relational results computed by evaluation of JSON_TABLE(). The master-detail table is linked by internally generated keys so that the column values in the master table are NOT repeatedly stored in detail tables. Indeed, the table index layout is the same as if certain parts of JSON objects were decomposed and stored relationally using E-R design [7]. The significance of table index is that it speeds up relational projection over a JSON object collection significantly. Unlike materialized view, table index is maintained synchronized with DML, multiple JSON_TABLE() expressions can be captured in one table index and maintained optimally by processing the input document once.

In summary, partial schema-aware indexing is very flexible because it builds secondary structures on top of the primary JSON object collection. Such secondary structures can be dropped and re-created without affecting the base JSON object collection. Users have the flexibility to drop and recreate index based on query workload and to accommodate schema evolution. **Therefore, it is more flexible to use partial schema to define index structures instead of using schema to define base table storage structures.**

6.2 Schema Agnostic Index Method

When there does not exist any partial schema in a JSON object collection, or when users can not anticipate query search patterns for a collection, we refer to this as the ad-hoc query use case. In such use case, JSON_EXISTS() is applied to a JSON object collection to search for existence of a certain JSON path with a member value satisfying range criteria. See Q1 of Table 2. One approach to index a JSON object collection in a schema agnostic way is to leverage the same idea as that of the XML path-value Index [23, 33]. A path-value XML Index provides the same index functionality as that of the vertical shredding

approach proposed in [9]. Although a vertical shredding approach can leverage SQL query over path-index relational tables directly to answer ad-hoc queries over XML or JSON object collections, it is inefficient to run the multiple self-join queries. Moreover, the resulting path-value index table size is much larger than the original document collection. It is typically at least 2 times of the original data size. This size problem gets worse if secondary indexes are included on the path-value index table. We propose to leverage inverted index layout [27,35] originated in the information retrieval community to build a schema agnostic JSON-aware inverted index to support ad-hoc queries. The advantage of an inverted index is that even though it indexes all keywords of all documents in a collection, the posting list for each keyword in the inverted index is highly compressed so that the total size of the inverted index is smaller than the size of original document collection [35]. A smaller index size improves I/O performance, generally the primary performance bottleneck in a DBMS. Each document in a document collection indexed by the inverted index is identified by an ordinal number as a DOCID. The DOCIDs of all documents containing the keyword are stored in a sorted manner with delta-compression as a posting list. Multi-predicate pre-sorted merge join (MPPSMJ) can be performed on the posting list in order to handle multi-keywords searches conjunctive query predicates [35, 41, 42] efficiently. We have leveraged and extended Oracle's built-in text index, an inverted index built inside RDBMS, which is similar to what is presented in [41] to not only index plain text documents but also index XML documents and JSON objects. By indexing XML tags and their hierarchical relationships, inverted index can be extended to support efficient processing of *containment queries* [34] which search for keywords within hierarchical paths in XML documents. Oracle builds XML full text index [42] by extending Oracle's text index to answer full text path containment queries for XQuery Full Text [32] expressions.

In the same way, we build a **JSON inverted index** by extending Oracle's text index to index JSON object member names, their hierarchical relationships and their content leaf data. JSON array elements are indexed with the parent array name containing them. Since we store JSON in ordinary SQL columns, we inherent the text index's ability to index textual content stored in VARCHAR, CLOB, RAW, or BLOB columns. Table 4 shows a DDL statement to create a JSON inverted index. Oracle text index internally assigns an ordinal number DOCID to each row of the table and maintains a bi-directional mapping between DOCID and ROWID of the table so that DOCIDs returned from inverted index lookup can return to the SQL engine as their corresponding ROWIDs and be subject to normal SQL processing.

```
Create index jidx on shoppingCart_tab (shoppingCart) index type context index (parameters
'JSON enable')
```

Table 4 – JSON inverted index

Unlike a standard text indexing tokenizer, the JSON inverted indexer operates on a JSON event stream derived from the underlying VARCHAR, RAW, CLOB, or BLOB column containing the JSON data. This event stream is generated by the JSON parser or one of the binary format decoders, as described in section 5.3. Concurrently, the JSON event stream consumer in the JSON indexer assigns each JSON object member name fetched from the event stream with an interval of starting and ending offset position. The interval of starting and ending offset position of an object member name is always contained by the interval of its parent object member name so that a hierarchical containment relationship between member names can be determined via testing the interval bounds. Leaf scalar data of a member is tokenized as keywords to facilitate full text search. Each keyword is assigned an offset position that is contained by the interval of the parent JSON object member name.

Each JSON object member name is indexed with a posting list consisting of the DOCIDs of JSON objects that contain the object member name and their list of intervals. Each keyword in the leaf data of a JSON object member content is indexed with a posting list consisting of the DOCIDs of JSON objects that contain the keyword and their list of offset positions. By performing an MPPSMJ join of the posting lists for both keywords and JSON object member names, we can facilitate full text query with JSON path navigation. The containment test between two path steps is done by testing containment of their intervals. The containment test between the keyword and the leaf path step is done by testing containment of the keyword offset position contained within the interval of the leaf path step.

By indexing both structures and data found in JSON objects, the JSON inverted index can be used to efficiently evaluate JSON_EXISTS() and JSON_TEXTCONTAINS() predicate used in SQL WHERE clause.

7. Performance Experiment

7.1 Experiment Setup

We use NOBENCH benchmark [9] data and run all queries Q1 to Q11 to evaluate query performance so as to evaluate the performance of our JSON native store and indexing strategy and how the strategy comparing with vertical JSON object shredding strategy used in paper [9]. We run performance experiments using Oracle RDBMS server that supports SQL/JSON that described in this paper. We create a table *NOBENCH_main* with a VARCHAR(4000) column *jobj* and load the table with JSON object instances generated using the criteria described in paper [9]. We create 3 functional indices and a JSON inverted index on the column *jobj* as shown in Table 5 with Q1-Q11 SQL/JSON queries shown in Table 6. We refer this as the 'Aggregated Native JSON Store' approach (**ANJS**) in this section. We have also implemented the vertical shredding approach of Argo/SQL for JSON objects as described in [9] using path-value relational table. We refer it as the 'Vertical Shredding JSON Store' (**VSJS**) in this section. All experiments are conducted on a PC running Linux kernel 2.6.18 with a 2.53 GHz Intel Xeon CPU and 6GB of main memory.

JSON OBJECT COLLECTION	*CREATE TABLE NOBENCH_MAIN(JOBJ VARCHAR2(4000))*
Functional Index	*create index j_get_str1 on NOBENCH_main(JSON_VALUE(jobj, '$.str1'));* *create index j_get_num on NOBENCH_main(JSON_VALUE(jobj, '$.num' RETURNING NUMBER));* *create index j_get_dyn1 on NOBENCH_main(JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER));*
JSON Inverted Index	*create index NOBENCH_idx on NOBENCH_main(jobj) indextype is ctxsys.context parameters('json enable ')*

Table 5 – NOBENCH Table and Index SetUp using Oracle RDBMS supporting SQL/JSON

Qry#	SQL/JSON Query
Q1	*SELECT JSON_VALUE(jobj, '$.str1') as str,* *JSON_VALUE(jobj, '$.num' \ RETURNING NUMBER) as num* *FROM nobench_main*
Q2	*SELECT JSON_VALUE(jobj, '$.nested_obj.str') as nested_str,* *JSON_VALUE(jobj, '$.nested_obj.num' RETURNING NUMBER) as nested_num* *FROM nobench_main*
Q3	*SELECT JSON_VALUE(jobj, '$.sparse_000') as sparse_xx0,* *JSON_VALUE(jobj, '$.sparse_009') as sparse_yy0* *FROM nobench_main* *WHERE* *JSON_EXISTS(jobj, '$.sparse_000') AND JSON_EXISTS(jobj, '$.sparse_009' j)*
Q4	*SELECT JSON_VALUE(jobj, '$.sparse_800') as sparse_800,* *JSON_VALUE(jobj, '$.sparse_999') as sparse_999* *FROM nobench_main* *WHERE* *JSON_EXISTS(jobj, '$.sparse_800')* *OR* *JSON_EXISTS(jobj, '$.sparse_999' j)*
Q5	*SELECT jobj* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.str1') = :1*
Q6	*SELECT jobj* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.num' RETURNING NUMBER) BETWEEN :1 AND :2*
Q7	*SELECT jobj* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER) BETWEEN :1 AND :2*
Q8	*SELECT jobj* *FROM nobench_main* *WHERE JSON_TEXTCONTAINS(jobj, '$.nested_arr', :1)*
Q9	*SELECT jobj* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.sparse_367') = :1*
Q10	*SELECT count(*)* *FROM nobench_main* *WHERE JSON_VALUE(jobj, '$.num' RETURNING NUMBER) BETWEEN 1 AND 4000* *GROUP BY JSON_VALUE(jobj, '$.thousandth')*
Q11	*SELECT left.jobj* *FROM nobench_main left* *INNER JOIN* *nobench_main right* *ON* *(JSON_VALUE(left_jobj, '$.nested_obj.str') =* *JSON_VALUE(right_jobj, '$.str1'))* *WHERE JSON_VALUE(left_jobj, '$.num' RETURNING NUMBER) BETWEEN :1 AND :2*

Table 6 –NOBENCH Queries using Oracle RDBMS supporting SQL/JSON

7.2 Performance Comparison of JSON query performance with and without JSON index

In this experiment, we use ANJS and compare Q1-Q11 performance with and without any JSON indices. Figure 5 shows the execution time ratio of running Q1-Q11 without having any indices and with indices. It represents the speed up due to using an index. Functional indices are used to speed up Q5, Q6, Q7, Q10, Q11 queries for dense static schema. A JSON inverted index is used to speed up Q3, Q4, Q8, Q9 queries for sparse dynamic schema. Other than Q1 and Q2, a JSON index speeds up queries significantly. Q1 and Q2 are queries to project out scalar values from JSON object without any predicates used in the WHERE clause so an index can't improve their performance.

7.3 Performance Comparison of JSON native store approach Versus vertical shredding store approach

We have implemented the Argo/3 approach of vertically shredding JSON objects and storing the resulting vertical relational table in the Oracle RDBMS. In this approach, there is a main path-value relational table similar to the table *argo_people_data* described in [9], holding objid, keystr and valstr. The valstr is indexed by a B+ tree index representing the *argo_people_str* table in [9]. An additional numeric B+ tree index is created on the valstr column for those string values that are valid numbers. This is the same as the Argo/3 table *argo_people_num* in [9]. Figure 6 shows the execution time ratio of running Q1-Q11 using VSJS with ANJS. It shows that ANJS with functional and inverted JSON indexes is faster than the VSJS approach.

The storage size of VSJS is much more than that of ANJS. For 50,000 JSON object instances, its total text size is 39MB. In ANJS, the *NOBENCH_main* table size is 39MB, the space for functional indices is 3.7MB and the JSON inverted index is 31MB. However, in VSJS, the base *argo_people_data* table size is 59MB. To facilitate string value search and number value range search, a B+ tree index to index *valstr* column of the *argo_people_data* table is created with an index size of 26MB. A B+ tree index to index valid numbers from valstr column of the *argo_people_data* table is created with an index size of 5MB. To facilitate keystr search, a B+ tree index on keystr column of the *argo_people_data* table is created with an index size of 39MB. Therefore, for VSJS, the total space consumption of *argo_people_data* with all of its secondary indices is 129.6MB which is 2.3 times more than the size of the base object collection of 34MB. In contrast with ANJS, the total space consumption of functional indices and inverted index is 34.7MB, which is 89% of the size of the base object collection size of 39MB. Figure 7 shows the size ratio comparison. A columnar store [1,30] of *argo_people_data* should be able to compress the storage size because all rows shredded from the same JSON object have the same object id column value, which can be compressed. A columnar store can also compress numeric and Boolean value columns that having many NULL values in Argo/1. Indeed the Argo/3 layout logically implements the column store and columnar store is known to be able to handle efficient storage and query of vertically shredded table. However, even if a columnar store is used, the reconstruction of full objects by assembling all the pieces stored in the vertical table is still expensive as demonstrated below.

Because any JSON store based on vertical shredding does **not** store the JSON object as a whole, it is costly to respond to common queries that retrieve the whole JSON object as their results. The store needs to run queries over the *argo_people_data* table to group all rows belonging to the same object id and then aggregate all columns of these rows to construct the full JSON object. This is mentioned in [9]: "Argo on the relational systems also suffers from more difficult object reconstruction as scale increases because it must access many (sometimes un-contiguous) rows when reconstructing matching objects". However, for ANJS, there is no need to construct the JSON object. Figure 8 shows the execution time ratio of VSJS versus ANJS for full JSON object retrieval query. The ANJS approach is 35 times faster than VSJS approach.

Therefore, VSJS is not a viable approach for storing JSON objects. Extending an inverted index to add JSON path support appears to be a more straightforward approach. Indeed, the inverted index architecture has much in common with columnar storage [1,30] for optimizing data layout to maximize query efficiency and has embraced all the index functionality of VSJS approach.

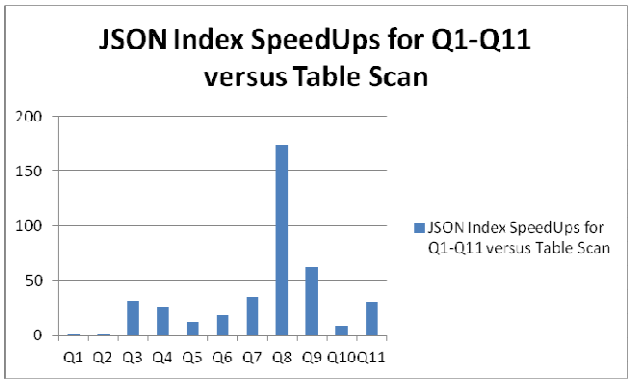

Figure 5 – JSON Index SpeedUps versus Table Scan

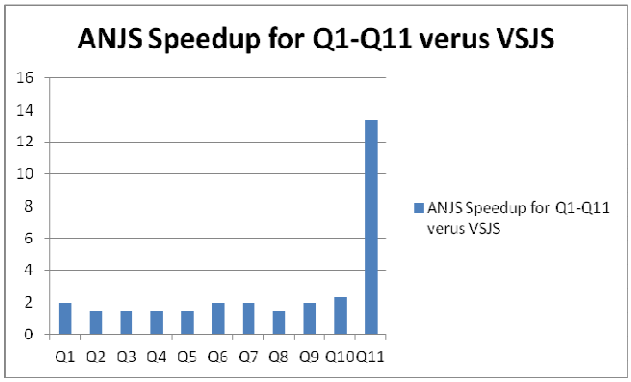

Figure 6 – ANJS SpeedUps for Q1-Q11 versus VSJS

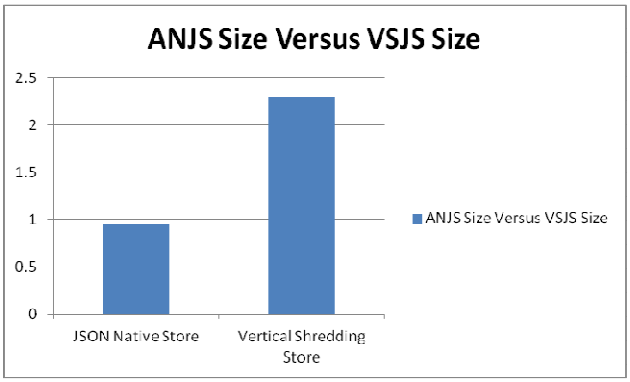

Figure 7 – ANJS Size versus VSJS Size

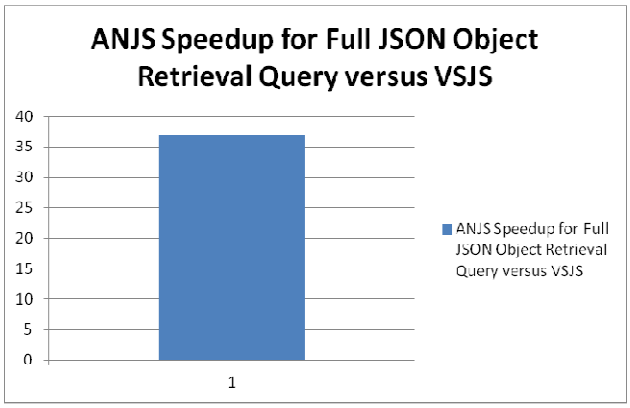

Figure 8 – ANJS Speedups for full JSON object retrieval Query versus VSJS

8. Future Work

JSON Rest API Access: A JSON object collection style of REST API can be supported to provide a simple API to access JSON persistence service in the RDBMS from a variety of imperative programming language environments that need to manipulate JSON data. A REST API will provide a No-SQL user experience to application developers; the underlying implementation can use the SQL/JSON operators described in this paper.

JSON inverted index supporting range value search: Compared with textual oriented documents, JSON objects are more data centric. Many JSON object members can be of data types such as numbers and dates where non-textual domain semantics are required for correct range semantics. Processing range expressions requires extending the JSON inverted index to index numbers, dates embedded in JSON objects.

JSON benchmark: NoBENCH [9] models the characteristics of JSON collection very well. However, to support OLTP and OLAP decision support workload for JSON object collections: we will work on benchmark that models multi-user CRUD operations on JSON object collections in high transaction context; we will also work on benchmark that models ad-hoc query and search over JSON object collection.

9. Conclusion

We are living in an interesting time where E.F Codd's relational model that relies on the existence of static data schema to store, query and index data is being challenged. This paper starts to address this challenge by extending the RDBMS to support schema-less development styles in conjunction with the classical schema-oriented development styles. The storage, query and indexing principles proposed in this paper are applicable not only to JSON and XML but to other semi-structured data management. In particular, the query principle positions SQL as a **S**et oriented **Q**uery **L**anguage so that the declarative power of SQL can be applied beyond plain relational data. Yet, the schema-less advantage of No-SQL is fully embraced in RDBMS with the three principles. The relational model has bought us tremendous success in structured data management. We hope that following the three principles of schema-less storage, query and indexing will help us build on this success continuously.

10. Acknowledgements

The authors like to express their thanks to following colleagues who have been in the working group to formulate SQL/JSON standard: Krishna Kulkarni, Fatma Ozcan, Jim Melton, Jan-Eike Michels, Hamid Pirahesh, and Fred Zemke along with authors of this paper. We thank Vikas Arora, Andrew Witkowski for management efforts of SQL/JSON development at Oracle.

11. REFERENCES

[1] D. J. Abadi, S. Madden, M. Ferreira: Integrating compression and execution in column-oriented database systems. SIGMOD Conference 2006: 671-682

[2] R. AgRAWal, A. Somani, Y. Xu: Storage and Querying of E-Commerce Data. VLDB 2001

[3] 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

[4] Beyer, K; Ercegovac, V; Gemulla, R; Balmin, A; Eltabakh, M; Kanne, C; Ozcan, F; Shekita, E: *Jaql: A Scripting Language for Large Scale Semistructured Data Analysis*

[5] 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

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

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

[8] Codd, E. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970)

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

[10] D.J. DeWitt: From 1 to1000 MIPS cs.wisc.edu/~dewitt/includes/passtalks/pass2009.pptx

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

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

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

[14] I.O. for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML)

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

[16] http://goessner.net/articles/JsonPath/

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

[18] K. Kunchithapadam, W. Zhang, A. Ganesh, N. Mukherjee: Oracle database filesystem. SIGMOD Conference 2011: 1149-1160

[19] Z. H. Liu, M. Krishnaprasad, H. J. Chang, V. Arora: XMLTABLE Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007

[20] Z. H. Liu, Chandrasekar,S; Baby T; Chang, H: Towards a physical XML independent XQuery/SQL/XML engine. PVLDB: 1356-1367 (2008)

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

[22] Z.H.Liu, H.J.Chang, B. Sthanikam: Efficient Support of XQuery Update Facility in XML Enabled RDBMS. ICDE 2012: 1394-1404

[23] Pal, S; Cseri, G; Schaller, O; Seeliger, L; Giakoumakis, V; Zolotov, V: *Indexing XML Data Stored in a Relational Database*. VLDB 2004: 1134-1145

[24] MongoDB : http://www.mongodb.org/

[25] 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

[26] R. Murthy, S. Banerjee: XML Schemas in Oracle XML DB. VLDB 2003 1009-1018

[27] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGRAW-Hill, New York, 1983.

[28] Shute, J; Vingralek, R; Samwel, B; Handy Ben; Menestrina, D; ellner, Stephan; Cieslewicz; J, Rae I; Stancescu, T; Apte, H: F1: A Distributed SQL Database That Scales. VLDB Conference 2013.

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

[30] Stonebraker, M.; Abadi, D; Batkin, A.;Chen,X.;Cherniack, M;Ferreira,M.;Lau, E.;Lin,A.;Madden,S; O'Neil, E.;O'Neil,P.;Rasin,A.;Tran,N.;Zdonik,S. *C-Store: A Column-oriented DBMS*. VLDB 2005: 553-564

[31] XQuery: http://www.w3.org/TR/xquery/

[32] XQuery Full Text: http://www.w3.org/TR/xpath-full-text-10/

[33] Yoshikawa, M; Amagasa, T; Shimura, T; Uemura, S: Xrel: A Path-Based Approach to Storage and Retreival of XML documents using Relational Databases

[34] C. Zhang, J.Naughton, D. DeWitt.J, Luo. Q, G.Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD Conference 2001: 425-436

[35] J. Zobel, A. Moffat: Inverted files for text search engines. ACM Computing. Surveys. 38(2): (2006).

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

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

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

[39] Qiao, L. etc: On Brewing Fresh Expresso: Linkedin's Distributed Data Serving Platform. SIGMOD 2013

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

[41] I. Rae, A. Halverson, J. Naughton: In-RDBMS Inverted Indexes revisited. ICDE 2014: 352-363

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