Chapter 2
JSON Data Management in RDBMS

Zhen Hua Liu
Oracle, USA

ABSTRACT

Being a simple semi-structured data model, JSON has been widely accepted as a simple way to store, query, modify, and exchange data among applications. In comparison with schema-oriented relational data model to store, query, and update application data, JSON data model has the advantage of being self-contained, free from schema evolution issues, and flexible enough to enable agile style development paradigm. Therefore, during the last 5 years, SQL/JSON standard has been established as foundation for managing JSON data in SQL standard and there have been JSON functionalities added into RDBMS products to support SQL/JSON standard to various degree. In this chapter, the authors will analyze the strength and weakness of using JSON as the data model to manage data for applications. For use cases where JSON data model is ideal, they present the design approaches to store, index, query, and update JSON in the kernel of RDBMS to support SQL/JSON standard defined operations effectively and efficiently.

JSON DATA MODEL AND ITS APPLICATION USE CASES

Why JSON Database?: Merits of Using JSON Data Model

Relational model (Codd, 1970) in RDBMS is based on 'schema first, data later' paradigm. Database users are required to design a relational schema before data can be stored. The schema is based on Entity-Relationship design practices (Chen, 1977) where an entity is composed of a set of attributes. Entities are related to each other through reference keys. Each entity is modeled as a table and every attribute of an entity becomes a column of the table. Each reference relationship is enforced as primary key and foreign key constraint. Foreign key columns of a table store reference keys to support reference relationship. To achieve update efficiency by avoiding data duplication, normalization rules (Fagin 1979) are followed so that there is no duplicated storage of the same data.

DOI: 10.4018/978-1-5225-8446-9.ch002

The issue with 'schema first, data later' paradigm is that when database applications evolve to add new attributes to an entity, the table for that entity must be altered to add new columns. This schema evolution issue becomes burden for database application developers who have to constantly request database administrators to evolve schema. Therefore, relational model is ideal for supporting highly structured data whose schema is relatively static.

In pure RDBMS, each column of a table is of a particular simple scalar datatype, such as integer, varchar, date, timestamp, etc. A complex type needs to be decomposed into multiple columns, each of which maps to a simple scalar datatype. An address type, for example, must be physically decomposed into three columns: street name, city name, zip code as the underlying storage columns. Furthermore, each column cannot be of an array type. An array type needs to be decomposed into two storage tables with reference relationship enforced by primary key and foreign key integrity constraint. Although object relational DBMS (ORDBMS) (Stonebraker, 1986) relaxes the simple datatype requirement by allowing complex datatype, including the array datatype, as a datatype of a column, the complex datatype definition, known as structured user defined type definition in SQL 99 (Melton, 2003), needs to be defined first so that columns of that complex datatype can store complex data. Therefore, schema evolution in the form of evolving structured user defined datatype definition with their implied physical storage structures remains. Furthermore, in ORDBMS, it is still true that each row of a table must have same number of columns and each column must be of same datatype whether it is complex type or not.

In contrast with the time when relational model and RDBMS were built, we are living in the big data age when there are variety of data to manage so that it is not practical to expect application users to design schema first before their application data are storable, indexable, queryable and manageable. In particular, for data that has loose structures, typically referred as semi-structured data, an alternative paradigm, known as 'data first, schema later', becomes more attractive. Two common semi-structured data models are XML and JSON. Native XML and JSON database systems are built based on the philosophy of 'data first, schema later' paradigm. MarkLogic and MongoDB are representatives of pure XML and JSON databases respectively. In particular, due to popularity of JavaScript, JSON, which represents the persistent data of JavaScript programming language, is a very simple semi-structured data and thus has gained its popularity during the last decade. JSON is a simple way to model an entity with flexible attributes. Each attribute can be of scalar type, object type, array type.

Compared with RDBMS, Native JSON database systems are based on document-object model instead of relational model. Native JSON DBMS supports concept of collection which is analogous to the concept of table in RDBMS. Each JSON document stored in a collection is analogous to a row stored in a table in RDBMS. Each JSON document is of document-object model. There can be variable number of attributes of various datatypes, including complex types, array types within each JSON document. Therefore, each JSON collection has flexible schema with variety of attributes that can be stored in each JSON document. Such schema flexibility in JSON database, in comparing with rigid schema requirement in RDBMS, really enables agile development style for application users who want a developer friendly instead of DBA friendly database system to manage their data. This trend has pushed RDBMS to embrace JSON capabilities as a schema-less development paradigm for application developers (Liu et al., 2014).

In summary, Table 1 shows the comparison between classical RDBMS and JSON native DBMS.

Table 2 shows an example of a JSON collection storing a set of JSON documents describing purchase orders.