Evolution of Phoenix as DocumentDB

Written by Viraj Jasani and Kadir Ozdemir

Basic Introduction of HBase and Phoenix:

Apache Phoenix is a relational database with Structured Query Language (SQL) interface built on top of Apache HBase, providing two key components. The first component includes a thick client with JDBC driver, SQL data type support, SQL language parser, Query planner with optimizer. The second component is integrated with server side HBase Coprocessors. Leveraging the powerful Coprocessor API of HBase, Phoenix utilizes these extension points for high performance integration. This integration enables Phoenix to offer structured metadata in the SQL model, optimize queries by leveraging push-down predicates and statistics collection, and implement strongly consistent secondary indexes, as well as hierarchical and tenant-specific views. Over the past couple of years, Apache Phoenix has also introduced some major feature enhancements including but not limited to uncovered indexes, partial indexes and view TTLs. The Salesforce Phoenix team has contributed many of the above mentioned features.

HBase stores rows of data in *tables*. Tables can be grouped into *namespaces*. A table can belong to only one namespace at a time. Tables are split into groups of lexicographically adjacent rows. These groups are called *regions*. By lexicographically adjacent we mean all rows in the table that sort between the region's start row key and end row key are stored in the same region. Regions are distributed across the cluster, hosted, and made available to clients by regionserver processes. Regions are the physical mechanism used to shard and distribute the write and query load across the fleet of regionservers. Regions are non-overlapping. A single row key belongs to exactly one region at any point in time. Together with the special *META table*, a table's regions effectively form a b-tree for the purposes of locating a row within a table. HBase is properly described as a distributed ordered tree.

Document Databases:

Let us define a Document Database. A document database is a type of NoSQL database designed to store, retrieve, and manage document-oriented information. Unlike traditional relational databases, which use tables and rows to structure data, document databases use a

more flexible and hierarchical data model where data is stored in documents. In a Document database, the data is stored in documents, which are typically JSON (JavaScript Object Notation) or XML (Extensible Markup Language) format. Each document is a self-contained unit of data that can include nested structures and arrays.

Binary JSON (BSON):

This document provides an introduction of a new datatype in Phoenix: BSON. BSON or Binary JSON is a Binary-Encoded serialization of JSON-like documents. BSON data type is specifically used for users to store, update and query part or whole of the BsonDocument in the most performant way without having to serialize/deserialize the document to/from binary format. Bson allows deserializing only part of the nested documents such that querying or indexing any fields within the nested structure becomes more efficient and performant as the deserialization happens at runtime. Any other document structure would require deserializing the binary into the document, and then perform the query.

BSONSpec: https://bsonspec.org/

JSON vs BSON and why do we need BSON?

JSON and BSON are closely related by design. BSON serves as a binary representation of JSON data, tailored with specialized extensions for wider application scenarios, and finely tuned for efficient data storage and traversal. Similar to JSON, BSON facilitates the embedding of objects and arrays.

One particular way in which BSON differs from JSON is in its support for some more advanced data types. For instance, JSON does not differentiate between integers (round numbers), and floating-point numbers (with decimal precision). BSON does distinguish between the two and store them in the corresponding BSON data type (e.g. BsonInt32 vs BsonDouble). Many server-side programming languages offer advanced numeric data types (standards include integer, regular precision floating point number i.e. “float”, double-precision floating point i.e. “double”, and boolean values), each with its own optimal usage for efficient mathematical operations.

Another key distinction between BSON and JSON is that BSON documents have the capability to include Date or Binary objects, which cannot be directly represented in pure JSON format. BSON also provides the ability to store and retrieve user defined Binary objects. Likewise, by integrating advanced data structures like Sets into BSON documents, we can significantly enhance the capabilities of Phoenix for storing, retrieving, and updating Binary, Sets, Lists, and Documents as nested or complex data types.

Moreover, JSON format is human as well as machine readable, whereas BSON format is only machine readable. Hence, as part of introducing BSON data type, we also need to provide a user interface such that users can provide human readable JSON as input for BSON datatype.

Grammar:

1) DDL Statement Grammar:

`BSON`

- **Definition:** The BSON-parsable Json to represent the BsonDocument.
- **Mapped to:** `org.bson.BsonDocument` Java object.

Example:

```
CREATE TABLE TABLE_NAME (PK1 VARCHAR NOT NULL, PK2 VARCHAR NOT NULL,
COL BSON CONSTRAINT pk PRIMARY KEY(PK1, PK2))
```

2) DML Statement Grammar:

```
{
 "<field-key>" : <field-value>
}
```

- <field-key> represents the String value of field keys for the key-value pairs stored in the BSON document.
- <field-value> represents the field value for the key-value pairs stored in the BSON document. Depending on the datatype of the field-value, it can be represented in different textual format. The supported data types for the field-value are provided here.

<field-value> data types with examples:

DataType	Textual format	Examples

String	sequence of characters	{
		"Color": "Gray"
		}

Number	Types of Number format:
	
	- Integer
	- Long
	- Double
	- Decimal
		{
		"Price": 75295.847,
		"Quantity": 10
		}

Boolean	Boolean values: true or false	{
		"ColorPresence": true
		}

Null	Null	{
		"Color": null
		}

Binary	Base64 Encoded binary values:
	
	"<binary-field-key>": {
	"$binary": {
	"base64": "<binary-field-value>",
	"subType": "<binary-sub-type>"
	}
	}
	
	BSON Binary sub-types are provided here: https://bsonspec.org/spec.html
		{
		"ColorBytes": {
		"$binary": {
		"base64": "QmxhY2s=",
		"subType": "00"
		}
		}
		}

List	List of field-values:
	
	"<list-field-key>": [
	<comma-separated-list-field-values>
	]
	
	field-values can be of any data type.
		{
		"Colors": [
		"Blue",
		"Red",
		123,
		true,
		null,
		"Orange"
		]
		}

Document	Document of field keys and values
	
	"<document-field-key>": {
	"<nested-field-key>":
	<nested-field-value>
	}
	
	nested-field-value can be of any data type.
		{
		"ColorDetails": {
		"ColorPresence": true,
		"Color": "Blue",
		"Quantity": 15
		}
		}

Set	Sets of field-values
	
	As Sets are not directly supported in JSON textual format, define “$set“ datatype:
	"<set-field-key>": {
	"$set": [
	<comma-separated-set-field-values>
	]
	}
	
	set field values should be of the same data type. e.g. set of string values or set of binary values etc.
		{
		"ColorSet": {
		"$set": [
		"Blue",
		"Red",
		"Orange"
		]
		}
		}

Date	Date type to support date representation.
	
	As Dates are not directly supported in JSON textual format, define "$date" datatype:
	
	"<date-field-key>": {
	"$date": "<data-time-format>"
	}
		{
		"PurchaseTime": {
		"$date":
		"2024-05-01T00:45:25.203Z"
		}
		}

Standard BSON Spec does not support Set Data type. The definition of Set as one of the complex data types within Phoenix supported Bson Data type is only specific to the use cases supported by Phoenix.

BsonNumber sub-types:

- BsonInt32: 4 bytes (32-bit signed integer, two's complement)
- BsonInt64: 8 bytes (64-bit signed integer, two's complement)
- BsonDouble: 8 bytes (64-bit IEEE 754-2008 binary floating point)
- BsonDecimal128: 16 bytes (128-bit IEEE 754-2008 decimal floating point)

UPSERT statement examples:

```
UPSERT INTO TABLE_NAME (PK1, PK2, COL) VALUES ('pk011', 'pk012',
   '{"Price":1234.123,"PurchaseTime":{"$date":"2024-04-01T00:00:20.2
   03Z"}}')

UPSERT INTO TABLE_NAME (PK1, PK2, COL) VALUES ('pk001', 'pk002',
   '{"Title":"Title
   Value","InPublication":true,"ColorBytes":{"$binary":{"base64":"Qm
   xhY2s=","subType":"00"}},"ISBN":"111-1111111111","Id2":101.01}')
```

SELECT statement returns the BSON Document object to JDBC API. The client can also retrieve the BSON column value as String, where String is the JSON representation of the BSON document.

Let’s deep dive into some BSON functions used with Phoenix SQL.

1. BSON Update Expression:

This section describes how to create new functions that can support updating Phoenix BSON document field key-value pairs from top fields or nested fields and store the BSON document back to Phoenix BSON column. The update expression can be used to update the document partially or completely. Though using an UPSERT statement with the new BSON document for the same column would replace the whole document completely, hence the update expression is mostly used to partially update the documents.

If the update expression has to update any nested fields, first we need to access the nested field using a document path, and then make the update to the given BSON field.

Top-level Fields:

A field or an attribute is said to be top level if it is not embedded within another field.

Nested Fields:

A field or an attribute is said to be nested if it is embedded within another field. To access a nested field, you use dereference operators:

[n] — for list elements
. (dot) — for document elements

Accessing list elements:

The dereference operator for a list element is [N], where n is the element number. List elements are zero-based, so [0] represents the first element in the list, [1] represents the second, and so on. Here are some examples:

MyList[0]
MyList[12]
MyList[5][11]

The element MyList[5] is itself a nested list. Therefore, MyList[5][11] refers to the 12th element in that list.

Accessing document elements:

The dereference operator for nested document element is . (a dot). Use a dot as a separator between elements in a document:

MyMap.nestedField
MyMap.nestedField.deeplyNestedField

The functions first retrieve the value of the field key by searching the field from top level of the BsonDocument. If the field key is not found, it follows `.` and `[ ]` notations for the nested fields.

Update expression Syntax:

```
{
  "$SET": { <field1>: <value1>, <field2>: <value2>, .... },
  "$UNSET": { <field1>: null, <field2>: null, ... },
  "$ADD": { <field1>: <value1>, <field2>: <value2>, .... },
  "$DELETE_FROM_SET": { <field1>: <value1>, <field2>: <value2>, ....}
}
```

An update expression comprises one or more clauses, each beginning with a SET, REMOVE, ADD, or DELETE keyword. Users can include these clauses in any order within the update expression. Nonetheless, each action keyword can only occur once.
Within each clause, there are one or more actions, delineated by commas, representing data modifications.

Action	Definition
$SET	Use the SET action in an update expression to add one or more fields to a BSON Document. If any of these fields already exists, they are overwritten by the new values.
$UNSET	Use the REMOVE action in an update expression to remove one or more fields from a BSON Document. To perform multiple REMOVE actions, separate them with commas.

$ADD	Use the ADD action in an update expression to add a new field and its values to a BSON document. If the field already exists, the behavior of ADD depends on the field's data type: * If the field is a number, and the value you are adding is also a number, the value is mathematically added to the existing field. * If the field is a set, and the value you are adding is also a set, the value is appended to the existing set. Definition of path and value in the context of the expression: * The path element is the document path to an field. The field must be either a Number or a set data type. * The value element is a number that we want to add to the field (for Number data types), or a set to append to the field (for set types).
$DELETE_FROM_SET	Use the DELETE_FROM_SET action in an update expression to remove one or more elements from a set. To perform multiple DELETE_FROM_SET actions, separate them with commas. Definition of path and subset in the context of the expression: * The path element is the document path to an field. The field must be a set data type. * The subset is one or more elements that you want to delete from the given path. Subset must be of set type.

Function Grammar:

Name: `BSON_UPDATE_EXPRESSION`

Arguments:

	Expression	DataType
1	Column Value	BSON
2	Update Expression	BSON

Definition: The function retrieves the BSON document for the given column, and performs updates to top-level and/or nested/complex fields of the document based on the update expression provided as part of the input. The value placeholders in the update expression are provided as third argument i.e. expression values. The expression value is the JSON string representing the BSON document for the expression values.

Return Type: BSON

2. BSON Condition Expression:

This section describes how to create new functions that can support accessing Phoenix BSON document field values from top fields or nested fields and evaluate conditional expressions. Condition Expression can be complex and it can include comparison of various scalar as well as nested/complex data types. Unless the client knows the datatype of the given document field key in advance, it is not feasible to compare the value at client side as part of the WHERE clause. The function is expected to include a BsonDocument of comparison values in addition to the condition expression.

Condition expression Syntax:

In the following syntax summary, an operand can be the following:

- A top-level field name
- A document path that references a nested field

The function supports two representations for Condition Expression:

1. SQL based condition expression, similar to condition statements defined in WHERE clause.
2. Tree based condition expression, similar to AST (Abstract Syntax Tree) used by various compilers while evaluating String based Condition Expression.

SQL based Condition Expression:

JSON to represent Condition Expression for SQL based conditional expression:

```
{
 "$EXPR": "<condition-expression-string>",
 "$VAL": {
   "<val-1>": <placeholder-val-1>,
   "<val-2>": <placeholder-val-2>,
   ....
 }
}
```

Here, the String value of the field “EXPR” represents the condition expression similar to conditional expression statements used in Java, Python or SQL. It uses Conditional Operators, Operands and Logical Operators with parentheses for higher precedence. The statement evaluation should be similar to SQL based condition evaluation.

Use these comparators to compare an operand against a range of values or an enumerated list of values:

- a = b – True if the field value of key “a” is equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.
- a <> b or a != b – True if the field value of key “a” is not equal to the field value represented by “b”. This compares value as well as datatype. If the values are same but the data types are different, True is returned. False is returned only if the field value of key “a” is equal to the field value represented by “b”, with same datatype.
- a < b – True if the field value of key “a” is less than the field value represented by “b”. For this comparison to be true, the data type must be matching.
- a <= b – True if the field value of key “a” is less than or equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.
- a > b – True if the field value of key “a” is greater than the field value represented by “b”. For this comparison to be true, the data type must be matching.

- a `>=` b – True if the field value of key “a” is greater than or equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.

Use the BETWEEN and IN keywords to compare an operand against a range of values or an enumerated list of values:

- a `BETWEEN b AND` c – True if a >= b and a <= c.
- a `IN (b, c, d)` – True if a = b OR a = c OR a = d.

Use document specific functions to verify field or value changes for top-level or nested fields in the document:

- `field_exists(a)` - True if a field with path “a” exists in the document. This path can represent top-level or nested fields.
- `field_not_exists(a)` - True if field with path “a” does not exist in the document. This path can represent top-level or nested fields.

Tree based Condition Expression:

JSON to represent Condition Expression for Tree based conditional expression:

```
{
 <logical-operator-1>: [
   <operand1>: {
     <comparison-operator-1>: <val-1>
   },
   <operand2>: {
     <comparison-operator-2>: <val-2>
   },
   ....
 ]
}
```

- Logical Operators include AND, OR, NOT. They can be represented with $ sign i.e. `"$and", "$or", "$not"`.

- Operands are Document field names in String.
- Comparison Operators include =, !=, <, >, <=, >= etc. They are represented with $ sign i.e. `"$eq", "$ne", "$lt", "$lte", "$gt", "$gte"`.
- Values for comparison are provided as BsonValue with special notation for Set, Binary and Date datatypes (which cannot be directly represented in JSON textual format).

	Condition Expression	Evaluation

1	{
	"a": {
	"$eq": "b"
	}
	}
		True if the field value of key “a” is equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.

2	{
	"a": {
	"$ne": "b"
	}
	}
		True if the field value of key “a” is not equal to the field value represented by “b”. This compares values as well as datatype. If the values are the same but the data types are different, True is returned. False is returned only if the field value of key “a” is equal to the field value represented by “b”, with the same datatype.

3	{
	"a": {
	"$lt": "b"
	}
	}
		True if the field value of key “a” is less than the field value represented by “b”. For this comparison to be true, the data type must be matching.

4	{
	"a": {
	"$lte": "b"
	}
	}
		True if the field value of key “a” is less than or equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.

5	{
	"a": {
	"$gt": "b"
	}
	}
		True if the field value of key “a” is greater than the field value represented by “b”. For this comparison to be true, the data type must be matching.

6	{
	"a": {
	"$gte": "b"
	}
	}
		True if the field value of key “a” is greater than or equal to the field value represented by “b”. For this comparison to be true, the data type must be matching.

7	{
	"a": {
	"$exists": true
	}
	}
		True if the field with path “a” exists in the document. This path can represent top-level or nested fields.

8	{
	"a": {
	"$exists": false
	}
	}
		True if the field with path “a” does not exist in the document. This path can represent top-level or nested fields.

Function Grammar:

Name: `BSON_CONDITION_EXPRESSION`

Arguments:

	Expression	DataType
1	Column Value	BSON
2	Condition Expression	BSON

Definition: The function evaluates the condition expression provided in the input. The value placeholders in the condition expression are provided as third argument i.e. expression values. The expression value is the JSON string representing the BSON document for the expression values.

Return Type: BOOLEAN

Examples:

SELECT * FROM TABLE_NAME WHERE BSON_CONDITION_EXPRESSION(COL, '**{"$and":[{"key1":{"$exists":true}},{"key2.key3[2]":{"$exists":false}}]}**')

UPSERT INTO TABLE_NAME VALUES (?,?) ON DUPLICATE KEY UPDATE COL = CASE WHEN BSON_CONDITION_EXPRESSION(COL, '**{"$EXPR":"PictureBinarySet = :PictureBinarySet AND NestedMap1.NestedMap2.NList[1] < :NList001","$VAL":{":NList001":12.22,":PictureBinarySet":{"$set":[{"$binary":{"base64":"MTIzX3JlYXIuanBn","subType":"00"}},{"$binary":{"base64":"MTIzYWJjX3JlYXIuanBn","subType":"00"}}]}}}**') THEN BSON_UPDATE_EXPRESSION(COL, '**{"$SET":{"Title":"Cycle_1234_new","Id":12345,"NestedMap1.ColorList":["Black","Silver"]},"$UNSET":{"IdS":null,"Id2":null,"NestedMap1.Title":null},"$ADD":{"AddedId":10,"NestedMap1.AddedId":10,"NestedMap1.NestedMap2.Id":-12345,"Pictures":{"$set":["xyz5@_rear.jpg","1235@_rear.jpg"]}}}**') ELSE COL END

3. BSON Value Function:

BSON_VALUE function is used to retrieve the value of the given document field for the given data type. It is essential for clients to identify the data type of the document field in advance because each SQL function needs to decode the value using one of the SQL supported data types.

Function Grammar:

Name: BSON_VALUE

Arguments:

	Expression	DataType
1	Column Value	BSON
2	Bson Field Key	The field key can represent any top level or nested fields within the document. The caller should use "." notation for accessing

		nested document elements and "[n]" notation for accessing nested array elements. Unlike nested fields, top level document fields do not require any additional character.
3	SQL Data Type	The data type that the client expects the value of the field to be converted to while returning the value.

Definition: The function returns the value of the given field key from the BSON Document. The client is expected to provide the data type that is used for decoding the value of the field key.

Return Type: Generic SQL Data Type (Depending on the third argument of the function, the data type conversion takes place)

Examples:

- BSON_VALUE(COL, 'topfield', 'DOUBLE')
- BSON_VALUE(COL, 'topfield.nestedfield1', 'VARCHAR')
- BSON_VALUE(COL, 'topfield.nestedfield[2]', 'INTEGER')

SELECT **BSON_VALUE(COL, 'topfield', 'DOUBLE')**, **BSON_VALUE(COL, 'topfield.nestedfield1', 'VARCHAR')** FROM TABLE_NAME;

Here, COL represents the column name of data type BSON.