SOLUTION BRIEF

SQL Abstraction for NoSQL Databases

cdata

SQL Abstraction for NoSQL Databases

The growth of NoSQL continues to accelerate as the industry is increasingly forced to develop new and more specialized data structures to deal with the explosion of application and device data. At the same time, new data products for BI, Analytics, Reporting, Data Warehousing, AI, and Machine Learning continue along a similar growth trajectory. Enabling interoperability between applications and data sources, each with a unique interface and value proposition, is a tremendous challenge.

The world of relational databases grappled with the need for consistent interfaces decades ago, and long since settled on standard means for communicating with data and application services. The relational database driver standards like JDBC, ODBC and ADO.NET. Database vendors, developers, and users have long relied on these common API standards, which has led to expansive ecosystems of compatible tools and familiarity to users of nearly all skill levels.

While many products lack broad or deep NoSQL support, the majority of applications that consume or generate data -- desktop tools, reporting servers, programming languages -- will implement robust support for generic database access through one or more of these driver interfaces.

These universal driver interfaces speak a common language – ANSI SQL. Over the last 40 years, SQL has established itself as the lingua franca of data access. While SQL does not marry perfectly with NoSQL structures, SQL-based drivers based on established data standards like ODBC, JDBC, and ADO.NET provide a powerful and ubiquitous bridge between NoSQL and modern and legacy applications alike.

The following solution brief highlights various techniques employed by CData Software (www.cdata.com) in providing SQL access to NoSQL data through their drivers. This paper discusses a variety of mapping and flattening techniques, and continues with vendor examples that highlight performance and usability differences between approaches.

Mapping SQL to NoSQL

Due to the flexibility of NoSQL, it is common for data structures to be returned as JSON objects, arrays, or any combination of hierarchical data. In order to work with these structures in common BI, Reporting, and ETL tools, the data objects typically need to be transformed into a tabular data format. The CData Drivers include several facilities for mapping or flattening these data structures to simplify integration with standard tooling.

These capabilities include:

- Extended Projection: requesting exactly the data you want from your tables.
- Horizontal Flattening: drilling down into embedded data (sub-documents and arrays).
- Vertical Flattening: treating embedded arrays of sub-documents as separate tables.
- Custom Schema Definitions: defining how the drivers view the NoSQL data.
- Client-Side JSON Functions: manipulating the data returned to perform client-side aggregation and transformation.

No single capability is appropriate for all NoSQL data sets, and there are costs and benefits associated with each approach. The techniques that users should employ to work with large or deeply-nested structures will be different than working with datasets which are less “object based”.

Furthermore, other vendors approach abstracting NoSQL to SQL differently. After outlining these NoSQL capabilities, this guide will outline other common approaches, including:

- Presenting all data in a table definition: exposing all sub-documents and embedded arrays as columns within a table definition.
- Creating parent-child relationships: separating sub-documents and embedded arrays as distinct tables and build parent-child relationships between the tables, as appropriate.

Ultimately, the best solution is the one that offers the most flexibility when it comes to NoSQL data interpretation, allowing users to tailor the solution to meet their any potential integration scenarios.

Sample Document

To demonstrate the mapping and flattening capabilities it is useful to include example data. The data structure below will be used in the examples that follow.

```
{
  "_id" : ObjectId("5780046cd5a397806c3dab38"),
  "address" : {
    "building" : "1007",
    "coord" : [-73.856077, 40.848447],
    "street" : "Morris Park Ave",
    "zipcode" : "10462"
  },
  "borough" : "Bronx",
  "cuisine" : "Bakery",
  "grades" : [
    {
      "date" : ISODate("2014-03-03T00:00:00Z"),
      "grade" : "A",
      "score" : 2
    }, {
      "date" : ISODate("2013-09-11T00:00:00Z"),
      "grade" : "A",
      "score" : 6
    }, {
      "date" : ISODate("2013-01-24T00:00:00Z"),
      "grade" : "A",
      "score" : 10
    }, {
      "date" : ISODate("2011-11-23T00:00:00Z"),
      "grade" : "A",
      "score" : 9
    }, {
      "date" : ISODate("2011-03-10T00:00:00Z"),
      "grade" : "B",
      "score" : 14
    }
  ],
  "name" : "Morris Park Bake Shop",
  "restaurant_id" : "30075445"
}
```

This data originated from a MongoDB database, however the tools and techniques presented here can be used with any supported NoSQL data source.

Extended Projection

In SQL, *projection* generally refers to collecting a subset of columns from a table for use in an operation. For example, issuing a SELECT query to pull a subset of columns from a table (`SELECT address, borough` etc.). *Extended projection* is the process of extracting data from non-tabular or hierarchical data sets.

Take the example of querying the included sample document from the root level. With NoSQL structures the data represented in these top-level 'columns' may be object arrays or aggregates. The default column list would include **_id**, **address**, **borough**, **cuisine**, **grades**, **name**, and **restaurant_id**. The **address** column is an example of an JSON object which would be returned as an aggregate with standard projection.

Through extended projection, the CData Drivers allow users to drill down into the sub-documents and embedded arrays (using dot notation) without having to issue complex joins or subqueries. By including **_id**, **address.street**, and **grades.0** as columns in the projection, users can expose a more atomic model of the data, accessing the traditionally exposed **_id** element as well as the **street** element of the **address** sub-document and the entire first element in the embedded **grades** array.

```
SELECT
  [_id],
  [address.street],
  [grades.0]
FROM restaurants;
```

The driver returns the value for each field that contains data. If the field does not exist in a given document, the driver returns *NULL*. This feature is useful whenever the structure of the data source is known and when users can issue distinct SQL Queries.

However, BI and ETL tools do not typically offer direct query access, particularly for reporting and visualization. In those instances, other flattening techniques should be used.

Horizontal Flattening

When a client does not have granular control of SQL Queries, flattening techniques should be used to map NoSQL to a tabular representation. The CData Drivers handle this process though Connection String settings.

The **Flatten Arrays** and **Flatten Objects** connection properties allow users to control how objects and array data is parsed in order to dynamically generate table schema for NoSQL data. These settings configure how data is horizontally flattened, creating a single schema for all of the documents (including embedded data) in a given table.

Horizontal flattening can be very useful for smaller data sets or those without deeply-nested object hierarchies. If the NoSQL data includes many sub-documents, large embedded arrays, or deeply nested data, then horizontal flattening may not be the best solution. In these cases, horizontal flattening may generate tables with too many columns, resulting in data sets that are overly granular and difficult to work with.

In the examples below, we display the expected results, based on various values for Flatten Arrays and Flatten Objects, for the following query:

```
SELECT *
FROM restaurants;
```

Default Behavior (FlattenObjects=False;)

Without any horizontal flattening, the Driver discovers seven columns for the table: **_id**, **address**, **borough**, **cuisine**, **grades**, **name**, and **restaurant_id**. Embedded data in the document is returned in a raw, aggregate form.

_id	address	borough	cuisine	grades	name	restaurant_id
5780046 …	{ "building" : "1007", "coord" : [-73.856077, 40.848447], "street" : …	Bronx	Bakery	[{"date" : ISODate("2014-03-03T00:00:00Z"), "grade" : "A", "score" : 2 }, { "date" : ISODate("2013-09-11T00:00:00Z"), …	Morris Park Bake Shop	30075445

FlattenObjects=True;

If a user sets **FlattenObjects** to "True", the number of columns expands as the embedded **address** sub-document is flattened. Without any changes to **FlattenArrays**, any embedded arrays or arrays of documents will be returned as aggregates:

_id	address.building	address.coord	address.street	address.zipcode	borough	...
5780046...	1007	[-73.856077, 40.848447]	Morris Park Ave	10462	Bronx	...

FlattenArrays=2;

The **FlattenArrays** property determines how many items in an embedded array of sub-documents are treated as individual columns. By setting **FlattenArrays** to "2", the driver extracts the first two items in the embedded arrays of a document.

_id	address	borough	cuisine	grades.0	grades.1	...
5780046...	{ "building" : "1007", "coord" : [-73.856077, 40.848447], "street" : "Morris Park Ave", "zipcode" : "10462" }	Bronx	Bakery	{ "date" : ISODate("2014-03-03T00:00:00Z"), "grade" : "A", "score" : 2 }	{ "date" : ISODate("2013-09-11T00:00:00Z"), "grade" : "A", "score" : 6 }	...

FlattenArrays=1;FlattenObjects=True;

With **FlattenArrays** set to "1" and **FlattenObjects** set to “True” the driver will extract the first item in the embedded arrays of a document and flatten any embedded sub-documents.

_id	address.building	address.coord.0	address.street	address.zipcode	...	grades.0.date	grades.0.grade	grades.0.score	...
57800...	1007	-73.856077	Morris Park Ave	10462	...	2014-03-03...	A	2	...

Any columns that are exposed from horizontal flattening are available for use in INSERT and UPDATE statements as well, allowing you to add or update individual fields within sub-documents and arrays. This approach works well for clients who know the structure of the data in advance and in data sets where the structure is not deeply nested.

Clearly, preconfiguring the driver prior to integration offers a richer experience when working with NoSQL data in tools and applications that expect a more traditional RDBMs Data Model.

Vertical Flattening

NoSQL databases frequently contain an array (or arrays) of sub-documents. While it is possible to drill down into these sub-documents using *horizontal flattening*, that approach presents problems when dealing with deeply nested data. Another approach for dealing with embedded arrays is to treat them as separate tables of data. This process is known as vertical flattening and doing so helps to build a relational model between different 'types' of documents in a hierarchy.

Using the included sample document, users could retrieve an array of grades as a separate table using the following query:

```
SELECT
  *
FROM [restaurants.grades];
```

This query would return the following data:

date	grade	score
2014-03-03T00:00:00Z	A	2
2013-09-11T00:00:00Z	A	6
2013-01-24T00:00:00Z	A	10
2011-11-23T00:00:00Z	A	9
2011-03-10T00:00:00Z	B	14

Users may also want to include information from the base restaurants table. This can be accomplished with a join. Flattened arrays can be joined with the root document.

The CData Drivers expect that the left part of the join will be the array document that will be vertically flattened. Set the S**upportEnhancedSQL** connection property to false to join nested documents.

For example:

```
SELECT
  [restaurants].[_id], [restaurants.grades].*
FROM
  [restaurants.grades]
JOIN
  [restaurants]
WHERE
  [restaurants].[name] = 'Morris Park Bake Shop' ;
```

This query would return the following:

_id	date	grade	score
5780046cd5a397806c3dab38	2014-03-03T00:00:00Z	A	2
5780046cd5a397806c3dab38	2013-09-11T00:00:00Z	A	6
5780046cd5a397806c3dab38	2013-01-24T00:00:00Z	A	10
5780046cd5a397806c3dab38	2011-11-23T00:00:00Z	A	9
5780046cd5a397806c3dab38	2011-03-10T00:00:00Z	B	14

Custom Schema Definitions

In order to create a relational facade on top of NoSQL data, a table schema must exist. The schema can be created dynamically through flattening via the Connection properties, or by pre-defining a schema. A pre-defined schema is another option for drilling down into data when it is impossible to maintain full control of the SQL queries constructed.

A Custom Schema gives users the ability to define the SQL structure that should be used when accessing the underlying NoSQL source. For example, using our sample data a user may want to define a table with **_id** (as the primary key), **name**, **zipcode**, and **latest_grade** (the first entry in the grades fields). The resulting schema would look like this:

```
<rsb:script xmlns:rsb="http://www.rssbus.com/ns/rsbscript/2">
<rsb:info title="StaticRestaurants"
          description="Custom Schema for the MongoDB restaurants data set.">
  <!-- Column definitions -->
  <attr name="id"           xs:type="int32" iskey="true" other:bsonpath="$._id" />
  <attr name="name"         xs:type="string" other:bsonpath="$.name"            />
  <attr name="zipcode"      xs:type="string" other:bsonpath="$.address.state"   />
  <attr name="latest_grade" xs:type="string" other:bsonpath="$.offices.grade"   />
</rsb:info>

<rsb:set attr="collection" value="companies" />
</rsb:script>
```

Schema files are saved to disk alongside the driver and referenced through the connection properties[1]. The driver will expose the defined tables based on the title attribute of *rsb:info*. Users can also query the data explicitly by using the title as the table name in a SQL query:

```
SELECT
  id, latest_grade
FROM
  StaticRestauraunts
```

By defining a schema, users gain granular control over data in ways not commonly supported in BI, reporting, and ETL tools, facilitating the use of visualization, transformation, and extraction features of those same tools. Custom Schemas also allow users to define different views of the data stored in a single table enabling users to take full advantage of the NoSQL database structure where a given table contains documents where relevant fields are differentiated by type definition.

Client-Side JSON Functions

NoSQL data structures are often represented as JSON structures. The CData Drivers support SQL functions for extracting data from JSON structures. A few examples of the JSON functions are highlighted below. For a complete list of supported functions reference the driver help documentation.

The following example uses the included sample document to extract values contained in the 'Students' table:

[1] Other techniques for embedding schema definitions are also available, but are beyond the scope of this paper.

```
{
  id: 123456,
  ...,
  grades: [
    { "grade": "A", "score": 96 },
    { "grade": "A", "score": 94 },
    { "grade": "A", "score": 92 },
    { "grade": "A", "score": 97 },
    { "grade": "B", "score": 84 }
  ],
  ...
}
```

JSON_EXTRACT

The JSON_EXTRACT function can extract individual values from a JSON object. The following query returns the values shown below based on the JSON path passed as the second argument to the function:

```
SELECT
  JSON_EXTRACT(grades,'[0].grade') AS Grade,
  JSON_EXTRACT(grades,'[0].score') AS Score
FROM Students;
```

This query returns the following data:

Grade	Score
A	96

JSON_SUM

The JSON_SUM function returns the sum of the numeric values of a JSON array within a JSON object. The following query returns the total of the values specified by the JSON path passed as the second argument to the function:

```
SELECT
  Name,
  JSON_SUM(score,'[x].score') AS TotalScore
FROM Students;
```

DOCUMENT

The DOCUMENT function can be used to retrieve the entire document as a JSON string. See the following query and its result as an example:

```
SELECT
  DOCUMENT(*)
FROM Students;
```

The query above returns each document in the table as a single string. E.x:

DOCUMENT
{ "_id" : ObjectId("5780046cd5a397806c3dab38"), "address" : { "building" : "1007", "coord" : [-73.856077, 40.848447], "street" : "Morris Park Ave", "zipcode" : "10462" }, "borough" : "Bronx", "cuisine" : "Bakery", "grades" : [{ "date" : ISODate("2014-03-03T00:00:00Z"), "grade" : "A", "score" : 2 }, { "date" : ISODate("2013-09-11T00:00:00Z"), "grade" : "A", "score" : 6 }, { "date" : ISODate("2013-01-24T00:00:00Z"), "grade" : "A", "score" : 10 }, { "date" : ISODate("2011-11-23T00:00:00Z"), "grade" : "A", "score" : 9 }, { "date" : ISODate("2011-03-10T00:00:00Z"), "grade" : "B", "score" : 14 }], "name" : "Morris Park Bake Shop", "restaurant_id" : "30075445" }

SQL to NoSQL Comparison

In order to choose the best approach for your integration, it is important to understand the various approaches to modeling NoSQL through SQL. CData provides a high degree of granularity and control when working with NoSQL data, however other vendors choose a different approach.

The following section will highlight the ways in which other popular vendors approach NoSQL mapping in the following scenarios:

- **Requesting All Table Data**: comparing the results of requesting all available data as a table (e.g. SELECT * FROM restaurants).
- **Embedded Arrays as Separate Tables**: comparing the results of sending JOIN queries to work with a table and the embedded data array(s).
- **Embedded Arrays & Sub-Documents as Table Elements**: comparing the results of working with embedded array(s) as elements within a table.

Requesting All Table Data

The most common way to retrieve all of the data in a traditional relational database table is to submit a `SELECT *` query. To compare techniques, we will display the results of a `SELECT * ...` query from various ODBC Drivers using their default connection properties.

There are 3 distinct approaches used by the top industry vendors.

The Drivers from Vendor 1 (V1) parse the grades array and address objects into separate tables linked by a parent-child relationship. The result of the select returns the following:

BOROUGH	RESTAURANT_ID	_ID	CUISINE	NAME
Bronx	30075445	5780046CD5A397806C3DAB38	Bakery	Morris Park Bake Shop
Brooklyn	30112340	5780046CD5A397806C3DAB39	Hamburgers	Wendy'S
Manhattan	30191841	5780046CD5A397806C3DAB3A	Irish	Dj Reynolds Pub And Restaurant
Brooklyn	40356018	5780046CD5A397806C3DAB3B	American	Riviera Caterer

The drivers from Vendor 2 (V2) similarly provide a subset of data when issuing a SELECT statement. The V2 drivers parse the grades and address.coord arrays as separate linked tables. The result is:

_id	address_ building	address_ street	address_ zipcode	borough	cuisine	name	restaurant _id
57800...	1007	Morris Park Ave	10462	Bronx	Bakery	Morris Park Bake Shop	30075445
57800...	469	Flatbush Avenue	11225	Brooklyn	Hamburgers	Wendy'S	30112340
57800...	351	West 57 Street	10019	Manhattan	Irish	Dj Reynolds Pub And ...	30191841
57800...	2780	Stillwell Avenue	11224	Brooklyn	American	Riviera Caterer	40356018

While creating separate tables distinguishes the different hierarchies in a NoSQL database, it can have significant performance and usability drawbacks when querying data from parent and child tables. The CData Drivers expose the most data by default, flattening the address object and drilling down to retrieve the fields available in the elements of the grades array.

_id	address.bui…	address.coord.0	address.coord.1	address.street	address.zip…	restaurant_id	grades.0.date	grades.0.grade	…
57800…	1007	-73.856…	40.8…	Morris Park Ave	10462	30075445	2014-03-03…	A	…
57800…	469	-73.961…	40.6…	Flatbush Avenue	11225	30112340	2014-12-30…	A	…
57800…	351	-73.985…	40.7…	West 57 Street	10019	30191841	2014-09-06…	A	…
57800…	2780	-73.982...	40.5…	Stillwell Avenue	11224	40356018	2014-06-10…	A	…

Embedded Arrays as Separate Tables

In the sample data, each document in the *restaurants* table contains an array of embedded documents in the *grades* element. This data represents the different grades a restaurant has received over time. As this is a hierarchy, the embedded documents capture the relationships between data by storing data in a single document structure.

By default, the drivers from Vendor 1 and 2 create a schema where the embedded documents are only recognized as separate tables (referred to as virtual tables and child tables respectively). They automatically create a table schema where the grades table shares a foreign key relationship with the *restaurants* table. The CData Drivers offer an additional level of control. They maintain the embedded documents, as elements within the original document, yet still allow users to treat the embedded values as separate tables.

Regardless of how the schemas are defined, all of the drivers are able to perform *JOIN* queries to retrieve related data between the tables.

Consider the following desired result-set:

restaurant_id	date	grade	score	P_id
30075445	2014-03-03T00:00:00.000Z	A	2	568c37b748ddf53c5ed98932
30075445	2013-09-11T00:00:00.000Z	A	6	568c37b748ddf53c5ed98932
30075445	2013-01-24T00:00:00.000Z	A	10	568c37b748ddf53c5ed98932
30075445	2011-11-23T00:00:00.000Z	A	9	568c37b748ddf53c5ed98932
30075445	2011-03-10T00:00:00.000Z	B	14	568c37b748ddf53c5ed98932

Queries and Performance

The queries required by the drivers to retrieve each grade as a separate row are relatively similar. Each driver uses an implicit JOIN to aggregate data. However, it is worth noting that the drivers from Vendors 1 and 2 require the use of a WHERE clause to identify the relationship between the two tables.

The CData Drivers on the other hand use vertical flattening (where child arrays are recognized as fields within the parent table, but can be treated as separate tables) to manage JOIN queries. The drivers from V1 and V2 treat the grades array as a separate table by default, meaning that data from both tables are pulled into memory and the drivers perform the JOIN client-side.

Driver	Time (seconds)	Query (returns all grades for approximately 10 million restaurants)
CData	252.9 (*+35% - +59%*)	**SELECT** [restaurants].[restaurant_id], [restaurants.grades].* **FROM**[restaurants.grades] **JOIN** [restaurants]

Vendor1	341.5	**SELECT** restaurants_grades.*, restaurants.restaurant_id
		FROM restaurants_grades, restaurants
		WHERE restaurants._ID = restaurants_grades.restaurants_id

Vendor2	401.2	**SELECT** restaurants_grades.*, restaurants.restaurant_id
		FROM restaurants_grades, restaurants
		WHERE restaurants._ID = restaurants_grades._id

Embedded Arrays & Sub-Documents as Table Elements

Document databases frequently contain embedded BSON/JSON objects and arrays as individual elements. The CData drivers are the only drivers that are capable of easily handling this type of structure. Consider the following result-set:

restaurant_id	grades.0.grade	grades.1.grade	grades.2.grade	grades.3.grade	grades.4.grade
123456780	A	A	A	A	A
123456781	B	B	B	B	B
123456782	C	C	C	C	C

With the CData drivers users can submit a free-form query to request the data as described above. The drivers use dot-notation to interpret requests for individual array objects and fields within sub-documents.

For example:

```
SELECT
  [restaurant_id], [grades.0.grade], [grades.1.grade], [grades.2.grade], [grades.3.grade], [grades.4.grade]
FROM
  [restaurants]
```

Other drivers interpret arrays of documents as separate tables. This means that users must perform JOIN queries in order to retrieve both *restaurant* and *grades* data. This interpretation means that there is no simple way to retrieve the grades for a given restaurant in a single row.

Conclusion

The CData approach to standardizing NoSQL integration builds upon the ubiquitous support for and familiarity with ODBC, JDBC, and similar standards. By wrapping data access in standards-compliant drivers, CData enables straightforward integration in any context where an RDBMS database would be used.

As Big Data continues to usher in a new era of analytics and cognitive computing, users are tasked with processing ever larger datasets. Data is now commonly aggregated from across multiple data sources and processing is federated across multiple nodes. As a result, the impact of seemingly small architectural differences can have a major impact on the performance of enterprise systems.

By offering granular control over NoSQL data interpretation, the CData Drivers exhibit a level of depth and flexibility unmatched by other products. These distinctions are supported by performance metrics executed when using the CData Drivers to process large amounts of NoSQL data.

While performance is beyond the scope of this paper, users should be cognizant of performance characteristics of any driver planned for production. For a complete picture of the data interpretation and performance features that set the CData Drivers apart, please refer to www.cdata.com/tech/bigdata/

For more information about CData Software, please visit the company website at: www.cdata.com.

For a complete list of current NoSQL Drivers, go to: www.cdata.com/drivers/

CData Software (www.cdata.com) is a leading provider of data access and connectivity solutions. We specialize in the development of Drivers and data access technologies for real-time access to on-line or on-premise applications, databases, and Web APIs. Our drivers are universally accessible, providing access to data through established data standards and application platforms such as ODBC, JDBC, ADO.NET, ODATA, SSIS, BizTalk, Excel, etc.

Our goal is to simplify the way you connect to data. We offer a straightforward approach to integration, with easy-to-use, data providers, drivers, and tools accessible from any technology capable of interacting with major database standards. This approach to integration allows businesses to realize the tremendous benefits and costs savings of integration while reducing complexity and expense.