IBM Information Management

IBM Informix® 12.10.xC2 Enhancements

Introducing NoSQL Capabilities

A Technical White Paper

IBM

Contents

Executive Summary

IBM® Informix® 12.10.xC2, in keeping up with the tradition of its prior versions, continues to evolve and enhance its leading database technology that is powerful, easy to use, and easy to manage. It continues to provide a high quality, high performing database engine for both OLTP and mixed OLTP/OLAP environments. Numerous enhancements were done to strengthen the existing server in the areas of security, administration, embeddability, cloud implementation, availability, application development, and supportability. Significant usability improvements were made available for time series data, and the ability to use external tables was added to the Informix Warehouse Accelerator (IWA) data mart. With the introduction of support for JSON as a native type and the ability to shard collections of documents, Informix has entered into the fast-paced, dynamic domain of NoSQL web and mobile application development, where engaging with customers has moved beyond merely recording transactions.

Introduction

IBM has introduced the ability to use the Informix and DB2 relational database management system (DBMS) to store NoSQL Javascript Object Notation (JSON) documents through the MongoDB clients. JSON has become the dominant format for information exchange in web and mobile applications, and MongoDB, which only exchanges information as JSON documents, has the largest market share of NoSQL document storage systems. Informix is more specifically an object-relational DBMS and has been since the mid-1990s. The existing ability to create new types and functions to work with them has made it possible to implement JSON and binary-JSON (BSON) types as first class citizens.

The domain of NoSQL database management systems is very broad. It encompasses NoSQL systems which do not manage relationships between sets of information at all and those not using the SQL language for such relationships (“no SQL” systems), and those with the capability to use both SQL and relational data, and non-relational functions for accessing structured information (“not-only SQL” systems). Informix is now one of the “not-only SQL” DBMS.

A large percentage of the NoSQL systems have been developed out of a need to work in the Web 2.0 environment. The JSON document format is a way to transfer object information in a way that is language neutral and is similar to XML in that respect. Language-neutral data transmission is crucial to being able to work in a web application environment, where it is common for an application to work with data from a variety of sources and pass data to and from software possibly written in different languages. The reduction in coordination required between application developers and database administrators enables more rapid development and deployment cycles.

Informix provides a unique platform where systems for engaging with customers can be easily integrated with systems for recording transactions.

The JSON Wire Listener

The JSON Wire Listener for Informix is the keystone to being able to access data as JSON documents. It uses the same communication protocol that MongoDB uses. The “wire” part of the name derives from this attribute; the protocol over the network (the wire) is the same. This means that applications can use the MongoDB client APIs with the listener in the same way that they use them with MongoDB. Applications written to use the MongoDB application programming interface can simply connect to the listener.

At the application layer, all data and all database operations come and go as JSON documents. The listener converts these JSON documents into SQL statements and server function calls as necessary. The server has JSON and BSON types defined and functions that operate on them. The BSON format is used to store the data.

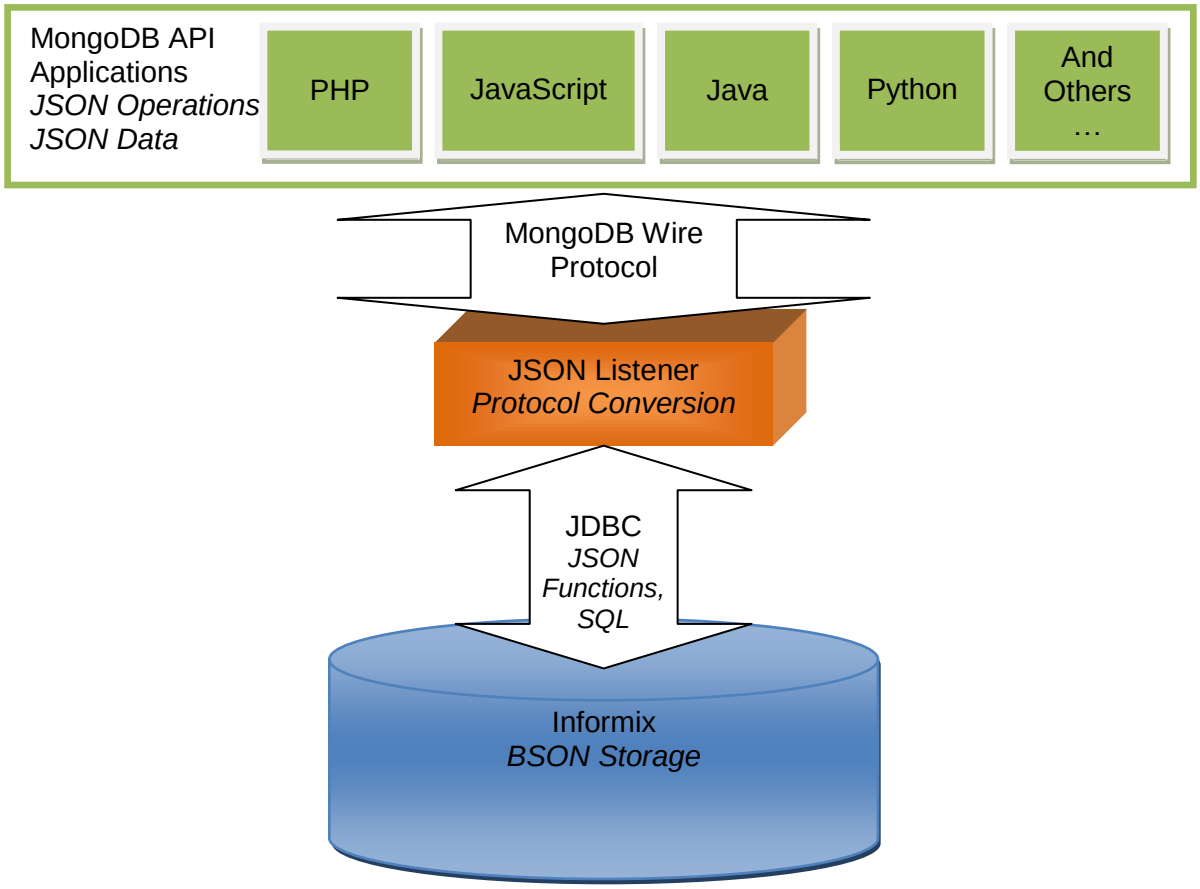

Figure 1. Architecture of the Informix NoSQL components.

The connection settings required by MongoDB are simply host and port number; the connection properties required by Informix also include server name, user name, and password. There are additional parameters to modify the behavior of the listener and the database server. You specify additional settings in a properties file in *<attribute>*=*<value>* pairs. The critical property that must be set is the connection URL. The URL is passed to the JDBC driver for establishing the connection between the listener and the database server. You can set additional attributes for the port number monitored by the listener, connection pooling, and miscellaneous settings.

A listener is associated with one server instance. One listener can service many application connections, and more than one listener can access a server at any time.

Connections are transparently pooled within the listener. Pooling connections simply means that connections that are closed between the application and the listener are not necessarily closed between the listener and the server. So, when the application requests a connection with the same attributes as one that has been used before, the listener can use one already established without having to go through authentication and resource allocation at the server. Connection pooling is required for performance reasons because in most NoSQL applications units of work are small, and each work unit gets its own database connection.

See the product documentation for a complete description of the listener settings available.

The name BSON stands for binary-JSON, but more accurately it is a JSON-like format because there are a couple of field types which do not map from one to the other. Also, the BSON type embeds length indicators which make traversal of documents more efficient.

JSON Example:

```
{ "greeting" : "Hello World" }
```

BSON Encoding:

```
\x1F\x00\x00\x00\x02greeting\x00\x0C\x00\x00\x00Hello
World\x00\x00

Length of encoding
Element type
Element name
Element value length
Element value
Encoding terminator
```

Collections are special types of tables within the server and the server maintains metadata about which symbol names are document collections and which are relational tables. If compression is available with your server license, you may compress collections.

You should use the MongoDB commands for adding or removing collections in order to ensure that the metadata stays consistent with the state of the collections. Similarly, MongoDB commands must be used when creating indexes.

NoSQL Quick Start Guide

There are three main components, the Informix server, the JSON wire listener, and a MongoDB client. Below is a brief description of how to install and configure these components.

Server

Installation Type
Select the Typical installation type. The Typical installation provides default server configuration options that reduce the complexity of installation and maintenance of the Informix server. This option installs all of the extensions that are required for using BSON and JSON types and also allows you to create a server instance. You can also choose the customized server settings for an instance and use the NoSQL extensions, but that is outside the scope of this document.

Number of Users
Select the range that includes the number of concurrent sessions that you expect to be typical for the server instance. This range does not change what is installed; it changes the configuration of the server instance created during the installation process.

Other Settings
The other options are unchanged from previous Informix releases. In most cases you can accept all default values.

JSON Listener

The wire listener is installed along with the Informix server, and started upon successful installation. The Java archive file is in the Informix installation directory, *bin* subdirectory, and the default configuration file is in the Informix installation directory, *etc* subdirectory. There is a user, *ifxjson*, created during the installation, and this user and password may be suitable for your use. You may want to change the **user** and **password** settings in the connection URL of the `jsonListener.properties` file, which is also in the *etc* directory. This URL is in the same format used by the Informix JDBC driver. There are also logging options for level of detail to log, and whether to put the log contents into a file or console output.

If the wire listener is not already running, you can start it by using the Informix Administration API or a system command. For example:

Command line argument

```
java -jar $INFORMIXDIR/bin/jsonListener.jar -start -config
$INFORMIXDIR/etc/jsonListener.properties
```

Informix SQL Admin API command：

```
EXECUTE FUNCTION task("start json listener");
```

Mongo Client

The installation instructions for the MongoDB server, shell, and client application programming interfaces (API) are located at the MongoDB open-source web site. To use the MongoDB clients to access the Informix server, point them to the host and port of the wire listener. Here is an example of starting the mongo shell:

```
./mongo --host frodo --port 27018
```

or

```
./mongo frodo:27018
```

Leveraging the NoSQL Listener to Access Relational Tables

You can utilize the same components used to access JSON documents to access rows stored in tables as though they were documents stored in collections. There are two ways to do this, using the $sql operator and referencing tables as you would collections. JSON is a format which allows for variable schemata; tables, obviously, have fixed schemata. It is easy to transform fixed schema data into a variable schema format. Transforming a variable schema JSON document into a fixed schema table row requires that the fields of the document map to the columns in the row.

If the Informix server, the JSON Listener, and the MongoDB client are not already installed and configured, please refer to the **Error! Reference source not found.**.

Mongo Operators

This functionality is intended to be very simple; read and write operations on existing tables are executed as though the table were a collection. The listener will examine the database and if the entity being accessed is a table, it will convert basic operations on that table to SQL, and convert the returned value(s) into a JSON document. The first access to an entity caches the name and type of that entity; so, the first access results in an additional call to the Informix server, but subsequent operations do not.

The listener is primarily for use with NoSQL databases and NoSQL databases contain at least one collection (else MongoDB will not make them permanent); so, the listener will not return a database from `show dbs` unless that database contains at least one collection. That does not prevent the database from being used; you can still connect to it through `use stores_demo`, for example.

Find

Using the **stores_demo** demonstration database, you can read all rows in the **customer** table with the following command:

```
db.customer.find()
```

Standard MongoDB operators can also be used[1]. Some examples are below:

[1] Not all operators, for example **$regex**, are currently supported.

```
db.customer.find({ customer_num: 119});  // return info on customer number 119
db.customer.find({ lname: "Shorter"}); // return customer with last name "Shorter"
db.customer.find({ lname: {$gt: "Shor", $lt: "T"} }  ); // AND conditions
db.state.find( {$or: [ {code: "AK"}, {code: "HI"} ] }, {sname: true} );
// OR conditions and projection list
```

Update

Update operations are also compatible with MongoDB. The following command changes the first name of the customer number 119 from 'Bob' to 'Robert':

```
db.customer.update( {customer_num: 119 }, {$set: { 'fname': 'Robert' } } )
```

Note that the operations are being performed on a relational table with a fixed set of columns; you cannot use an **UPDATE** statement to add or remove columns as you can use one to add or remove fields in a document.

Insert

Insert operations where the named entity is a table will attempt to add a row to the table; operations where the table does not exist or where the entity is a collection will result in a row being added to a collection. If the **INSERT** is to a table, it can fail for all the same reasons an SQL **INSERT** would fail, including non-existent columns or integrity constraints.

If the following command is run while connected to a **stores_demo** database, it will result in a new customer record being created in the customer table.

```
db.customer.insert(
{
    "fname" : "H. G.",
    "lname" : "Wells",
    "company" : "Mars, Inc.",
    "address1" : "4th Rock",
    "city" : "Canal 5",
    "state" : "Olympus Mons",
    "zipcode" : "90211",
    "phone" : "602-867-5309"
})
```

The same command run when there is no **customer** *table* results in a document with those values existing in the **customer** *collection*.

Remove

Remove operations work either on existing tables or on collections in the same way that `insert()` operations do with the obvious exception that a removal of a document does not implicitly result in a collection being created. The following command results in the row that was added above being deleted.

```
db.customer.remove( {state : "Olympus Mons"  })
```

SQL Command Pass Through

SQL is a language designed to operate on sets, and there are set operations and expressions, like joins, which cannot currently be expressed in single MongoDB operations. The `$sql` operator allows for the execution of SQL commands within the Informix database engine. The find() or findOne() methods can be used to execute data definition language. The result of the execution is returned as a document containing a single field "n" indicating the number of rows affected.

SQL operations are disabled by default; this is partly to prevent unauthorized access. It is enabled by setting the **security.sql.passthrough** property in the jsonListener.properties file to true.

```
listener.port=27018
url=jdbc:informix-sqli://frodo:27849/sysmaster:INFORMIXSERVER=nosql1210;
       USER=<username>;PASSWORD=<password>
security.sql.passthrough=true
```

The following examples demonstrate how you can use the $sql operator.
Create Table

```
mongos> db.getCollection("$sql").find({ "$sql": "create table foo (c1 int)" })
{ "n" : 0 }
```

Drop Table

```
mongos> db.getCollection("$sql").find({ $query: { "$sql": "drop table foo" }})
{ "n" : 0 }
```

Queries
The SQL language does not require that all columns be named, or named uniquely, for example:

```
select
        1+1,
        c.customer_num,
        o.customer_num
from
        customer c, orders o
where
        c.customer_num = o.customer_num
```

This kind of query will cause problems in converting the column names into JSON field names because every field in a document has to have a unique name. These problems can be avoided by providing aliases for any unnamed or ambiguously named columns, such as:

```
select
        1+1 as foo,
        c.customer_num,
        o.customer_num as order_customer
from
        customer c, orders o
where
        c.customer_num = o.customer_num
```

Join Select
There is currently no way to express join relationships between documents in a single MongoDB API call. However, you can use the ability to execute arbitrary SQL to generate results of join operations. For instance, here is a query to count the number of orders a customer has placed; it uses an outer join to include the customers who have placed no orders.

```
db.getCollection("$sql").find({ "$sql":
        "select
            c.customer_num,
            o.customer_num as order_cust,
            count(order_num) as order_count
        from
            customer c left outer join orders o
            on
            c.customer_num = o.customer_num
        group by 1, 2
        order by 2" })
```

The result of this query looks like:

```
…
{ "customer_num" : 113, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 114, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 101, "order_cust" : 101, "order_count" : 1 }
{ "customer_num" : 104, "order_cust" : 104, "order_count" : 4 }
{ "customer_num" : 106, "order_cust" : 106, "order_count" : 2 }
…
```

In general, there are a couple of ways to deal with null values. From the NoSQL perspective, if there is no value for an attribute, it is common to not create a document field created for that attribute. From the SQL perspective, it might be known that an object has an attribute even if the value of the attribute is not known; in which case, it is better to include the field in the document. The latter solution is chosen by the listener because it more closely preserves the meaning stored in the table. This behavior is also compatible with, for instance, the MongoDB C# driver.

Delete

The following command will delete records of customer calls more than 5 years old.

```
mongos> db.getCollection("$sql").findOne({ "$sql": "
delete
from cust_calls
where
    (call_dtime + interval(5) year to year) < current" })
```

Result:

```
{ "n" : 7 }
```

The returned document indicates that seven rows were changed (deleted) as a result of the operation.

Update

The following command increases the price of items manufactured by Hero by 10% and returns one document that indicates that twelve records were updated.

```
mongos> db.getCollection("$sql").findOne({ "$sql": "
update stock
set unit_price = unit_price * 1.10
where manu_code = 'HRO'" })
```

Result:

```
{ "n" : 12 }
```

Horizontal Scaling

Design

Horizontal scaling and sharding are used in the same context. Horizontal scaling refers to increasing workload capacity by distributing the workload across additional physical machines, and sharding means to split your data into distinct subsets, where each subset is a shard. Sharding data is a way to achieve horizontal scalability. Data can be replicated within a shard node, but data storage is not shared between nodes. In contrast, vertical scaling refers to increasing the capacity of the machines storing the data.

Distributing workload within the domain of traditional relational systems can be done with either the same data replicated, or shared, at different nodes, or different nodes sharing none of the data, but in either case, when the database server itself does not have this capability some form of middle-tier application is required to coordinate the requests between the end-user application and the database servers. Sharding is an example of the shared-nothing scenario. It effectively pushes the functionality of mapping of work requests out to servers and reduction (or aggregation) of partial results down into the database server itself. Implementing sharding effectively is still not a trivial task, but reducing the layers within the application stack can make it simpler. In addition, the metadata about which shards may contain the information requested does not have to be maintained outside of the database system.

Fragmented (partitioned) tables are tables which have their rows divided between different physical storage devices, where each storage device contains a distinct subset of a table's rows. The purpose of fragmenting tables is to relieve performance bottlenecks due to the physical limitations of a single disk or other storage device. Sharding is similar in concept except that instead of only distributing the storage-layer read and write operations, the workload on the CPU and memory is also distributed. There is a tradeoff for this architecture in that it requires network communication between the node receiving the work request and the other nodes containing some portion of the data needed to produce a result. Communication between hardware nodes is orders of magnitude slower than processing done within a hardware node. The rate of requests (operations per second) must be very high and the size of the data to be processed must be very large in order to compensate for the cost of having to coordinate the request between different nodes.

A shard of data is associated with a server node. A server node can be a single server, or a replica set. In the Informix sharding solution, every server node knows the distribution of sharded collections; so, applications can connect to any node to perform operations on sharded collections. Collections that are not sharded can only be accessed from the node hosting the collection. In the illustration below, application A can access the Foo and Bar collections (dotted green line). An application connecting through another listener to node 2 would be able to access all of the Foo collection, but would not be able to access the Bar collection. If a shard key is not provided (blue dashed line) for a query, then the receiving node will broadcast the request to all nodes. All Informix servers participating in a shard cluster are aware of which nodes contain what portions of a sharded collection. So, if it is possible for the server to determine that only a subset of nodes might contain the relevant data, operations limited to a subset of the documents are directed to only those nodes holding the relevant shards. Again in the illustration below, if a query contains a shard key (red solid line) condition indicating that the document(s) belong in shard 3, then node 1 would only pass the operation to node 3.

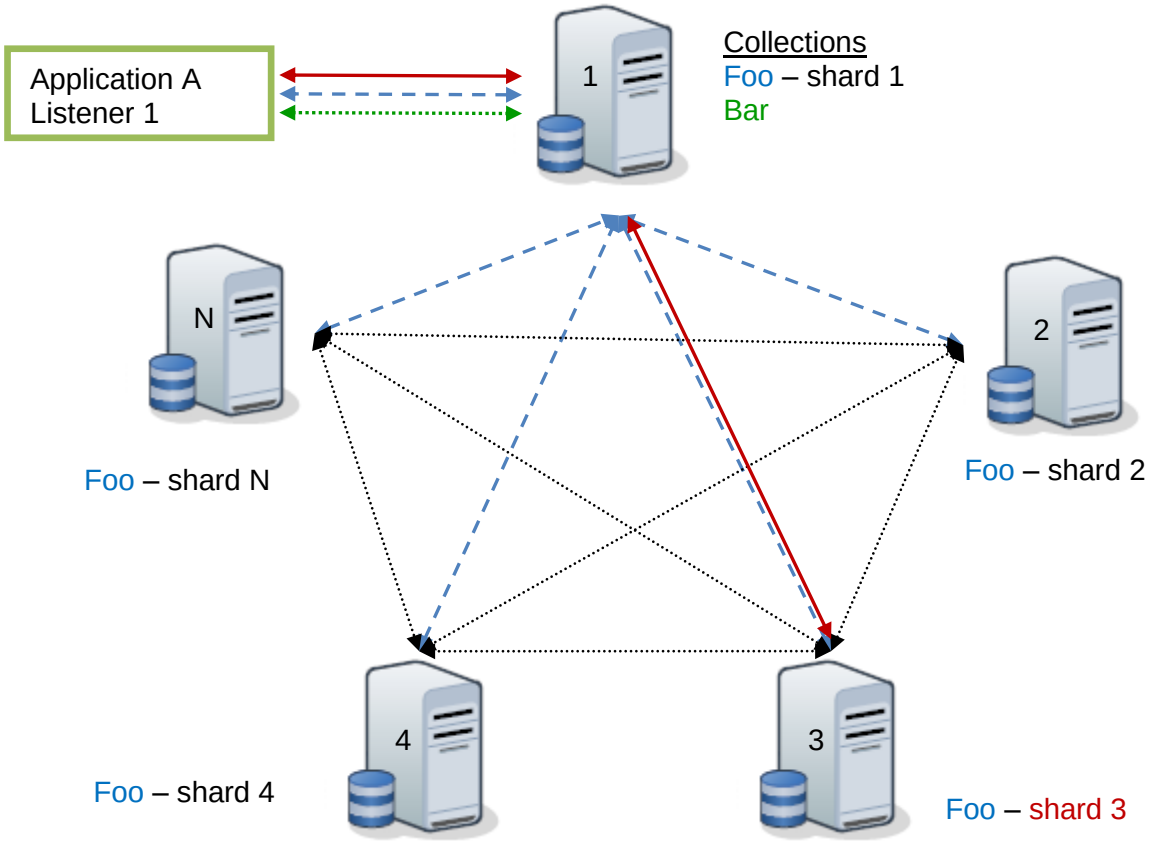

Figure 2. Prototype of Informix shard cluster.

Sharding in Practice

Sharding adds complexity to your database landscape and necessarily adds some latency to each request because of the additional network path. Informix scales very well vertically already; so, the need to shard is reduced, but the price of hardware is variable over time and not linear with respect to the ratio of price to performance. Some scenarios will exist where the price of an additional system is less than the price of a system with faster, or more, CPUs, memory, or disk storage, for the same workload capacity gain. In general, there have to be a lot of users accessing a lot of data in order for sharding to provide benefits. If the data set is not very large, the database server can cache a significant portion of it in memory, and read operations of in-memory data are very fast. If there are a limited set of users, for instance employees accessing an internal system, then it would be unusual for there to be enough requests per second to justify sharding. A high volume of write operations can make sharding more desirable because more write operations closely correspond to disk operations, but read operations can often be fulfilled by data already in memory.

Sharding Quick Start Guide

Setup

You need to specify the trusted hosts for each database server that is to be part of the shard cluster. You do this with the SQL administration API task() function with the add trustedhost argument and customized host values.

```
EXECUTE FUNCTION task("cdr add
        trustedhost","myhost1, myhost1@ibm.com, myhost2, myhost2@ibm.com");
```

Sharding needs to be enabled in the wire listener; set the *shard.enable* parameter to *true* in the *jsonListener.properties* file. If you change the value of this parameter, you must restart the wire listener for the change to take effect. If you have *shard.enable* set to false in the wire listener properties file, the queries on sharded collections only reflect data stored on the local database server.

Administer sharding from MongoDB shell

You can run commands related to shard administration using the MongoDB shell. Connect the MongoDB shell to the wire listener for Informix. If the MongoDB shell is on the same machine as Informix and the wire listener, and you are using the installed default settings:

```
% mongo
```

If the MongoDB shell is on a different machine, or you have changed the listener's port number:

```
% mongo --host <hostmachine> --port <portnum>
```

Add shard servers

Use the addShard command to add new servers to the shard cluster. The addShard command adds new servers to the shard cluster and new servers to existing, hash-based sharded collections. This automatically rebalances your data in the background.

Examples:

```
sh.addShard("myhost1:port1")
db.runCommand({"addShard":"myhost1:port1"})
db.runCommand({"addShard":["myhost1:port1","myhost2:port2", "myhost3:port3", …]})
```

List shard servers

Use the listShards command list shard servers.

```
mongos> db.runCommand({listShards:1})
{"serverUsed" : "myhost1/9.25.152.51:9201",
"shards" : [
        {"_id" : "g_ol_informix1210_1", "host" : "myhost1:9201"},
```

```
        {"_id" : "g_ol_informix1210_2", "host" : "myhost2:9202"}],
"ok" : 1
}
```

Shard collections

You can use the shardCollection command to enable collection sharding and data distribution across the servers in the cluster. You can shard a collection by hash or by expression. This example shards the collection named mycollection1 in the mydb database using a hash algorithm on the _id field in the documents.

```
sh.shardCollection("mydb.mycollection1", {"_id":"hashed"})
```

To shard by expression, you provide the expressions that determine which documents are stored on each shard server. The following example shards the collection named mycollection2 in the mydb database using the state field as the shard key. The data are distributed across servers as defined in the expressions.

```
db.runCommand({
        "shardCollection":"mydb.mycollection2", "key":{"state":1},
        "expressions":{
                "g_server_1":" in ('KS','MO')",
                "g_server_2":"in ('TX','OK')",
                "g_server_3":"remainder"}
        })
```

Scaling out

As your data and capacity requirements grow, you can scale out by adding more shard servers using the *addShard* command. *AddShard* automatically adds new shard servers to any existing hashed shard rules. To add new shard servers to expression shard rules, use the *changeShardCollection* command. The *changeShardCollection* command is only supported for expression-based sharding.

Change expression shard rules

You can use the changeShardCollection command to change the expressions used for expression-based sharding. This includes adding or removing shard servers from the sharded collection.

Example:

```
db.runCommand({"changeShardCollection":"mydb.mycollection2", "expressions":{
        "g_server_1":"in ('KS','MO')",
        "g_server_2":"in ('TX','OK')",
        "g_server_3":"in ('WA','OR')",
        "g_server_4":"in ('CA', 'NM','NV')",
        "g_server_5":"remainder"
}})
```

Broadly Applicable and Warehouse Accelerator Enhancements

Informix 12.10.xC2 incorporates enhancements in all aspects of the server including installation, migration, administration, performance, security, embeddability, application development, cloud, and availability. Significant performance and usability improvements were implemented for the sensor data management hosted on Informix Warehouse Accelerator (IWA) data marts.

In the area of performance, some of the many enhancements include: in-place conversion of data types from SERIAL to SERIAL8 and BIGSERIAL, SERIAL8 to BIGSERIAL and BIGSERIAL to SERIAL8 using the ALTER TABLE. Earlier, such data type conversions used slow alter operations. In-place alter operations require less space than copy alter operations and makes the table available to the other sessions faster. Other enhancements also include improving query performance for statements that use the LAST COMMITTED option of the Committed Read isolation level by enabling read ahead and light scans. This improvement aids warehousing applications by improving the performance of complex analytical queries.

Enhancements to sensor data include enabling replication of time series data to all types of high-availability cluster nodes hosted in read-only mode including the previously supported High-Availability Data Replication (HDR), shared-disk secondary, and remote stand-alone secondary clusters. The usability of TimeSeries data has been improved by allowing the use of TimeSeries columns in the ORDER BY clause of an SQL statement. Performance improvements in sensor data handling include use of *aggregation of an interval of time series data, parallelizing queries on virtual tables created using fragment by expression, tunable data flushing using the time series loader, and accelerating queries on time series data in IWA data marts.*

The following section summarizes the enhancements done in IWA in this release. For additional information and the complete list and explanation of all the other enhancements, certification and compliance, refer to the release notes and the corresponding release documentation.

Accelerating queries on partitioned time series data in IWA
Informix TimeSeries data type tables contain time stamped data based on a time series. Time series data sets are usually very massive. The proprietary Informix TimeSeries data type provides a superior solution with regards to performance and disk space consumption and management when working with time series data.

The TimeSeries data type is not directly supported by IWA, but you can create a Virtual Table Interface (VTI) of the TimeSeries data to project it as a simple relational representation.

A typical warehousing schema of a database containing time series data consists of a fact table (containing a number of time stamps/series), and dimension tables, forming a star or snowflake

schema. When querying such a schema, the TS virtual table (TSVT) gets joined with dimension tables.

In 12.10.xC1, the IWA already supported creating data marts with fact tables containing time series data using the VTI. When TimeSeries data is projected to a VTI, the number of rows expand exponentially, which could make some of the queries on the time series data that have multiple joins by virtue of star schema inefficient. Also, typically time series data in the previous versions of IWA was not partitioned. Unpartitioned data could slow the load speed and query execution time, and to refresh the data you must reload the complete virtual table. The new 12.10.xC2 release alleviates this bottleneck by enabling the partitioning of the time series data based on time intervals in the VTI in the fact tables. You can now define smaller virtual partitions (via VTI) that can be used to either quickly refresh the data in part of the IWA data mart or continuously refresh the data.

The partitioning on the virtual table is enabled through TimeSeries calendar properties. This is done using the following steps:

- Create the virtual table using the TSCreateVirtualTab() procedure and enable the TSVTMode parameter by including the TS_VTI_SCAN_DISCREET setting.
- Define virtual partitions by assigning a calendar.
- To virtually partition time series data in an IWA data mart VTI, a time series calendar must be created and assigned to the time series virtual table using the ifx_TSDW_setCalendar() routine. The calendar index identifies a virtual partition. The first partition is numbered 0.

  Example:

```
-- Create a calendar for the partition time range
INSERT INTO calendartable (c_name, c_calendar)
VALUES ('2013monthly',
        'startdate(2013-01-01 00:00:00.00000),
       pattstart(2013-01-01 00:00:00.00000),
pattern({1 on},month)');

-- Partition the data as per the defined calendar
EXECUTE FUNCTION ifx_TSDW_setCalendar('demo_dwa', 'demo_mart',
'informix', 'ts_data_v', '2013monthly');
```

  Where 'demo_dwa' is the accelerator name, 'demo_mart' is the IWA data mart name, 'informix' is the owner of the table, 'ts_data_v' is the virtual table name and '2013monthly' is the time series calendar.

  In the above example the calendar index `0` identifies the virtual partition for January 2013, the calendar index `1` identifies February 2013, and so on.

- Refresh time series virtual partition
  After you define virtual partitions and load the data mart, you can refresh the time series data for a single virtual partition by running the ifx_TSDW_updatePartition() routine.

  Example:

```
ifx_TSDW_updatePartition('demo_dwa', 'datamart_name', 'informix', 'ts_data_v', 2);
```

The example above will refresh data for the month of March 2010 as per the calendar definition.

This enhancement increases the system availability and makes continuous data refresh of TimeSeries data to IWA faster. It also saves the overall time by accelerating analytical queries while making the IWA data mart administration simpler, thus increasing the business value.

Loading data from external tables into IWA data marts

An external table is a data file that is not managed by an Informix database server. This could be flat files which conform to a well defined record format that the database can translate. The definition of the external table includes data-formatting type, external data description fields, and global parameters. To map external data to internal data, the database server views the external data as an external table. Treating the external data as a table provides a powerful method for moving data into or out of the database and for specifying transformations of the data. External tables also let you query and write data as though the data where in Informix permanent table.

You can define an external table by using the `CREATE EXTERNAL TABLE` statement.

Example:

```
CREATE EXTERNAL TABLE MyEmpTab (
    Name     CHAR(18)    EXTERNAL CHAR(18),
    Hiredate DATE        EXTERNAL CHAR(10),
    Address  VARCHAR(40) EXTERNAL CHAR(40),
    Empno    INTEGER     EXTERNAL CHAR(6))
USING (
    FORMAT 'FIXED',
    DATAFILES("DISK:/anr/emp.fix")
);
```

The above statement defines the external data format and creates the external table definition with data referenced from the disk file named /anr/emp.fix. The file can also come from a pipe (tape drive or direct network connection).

Prior to this release, users had to load the external data from the flat files into the Informix database and then perform the analysis there. With 12.10.xC2, by enabling the external tables support, users could consider loading the external data directly into an IWA data mart for further analysis.

Enabling this key feature of loading data from external tables directly to IWA data marts provides the following benefits:

- Transferring data from any source across platforms in an ASCII-delimited file to IWA
- Performing parallel standard INSERT operations
- Using named pipes for loading data from storage devices and direct network connections
- Maintaining a record of load statistics during the run
- Performing high-speed and data-checking data load
- Mixing external and permanent tables in the IWA data mart
- No loading/building of tables/indexes required

- Saving data preparation time through minimal setup
- Saving storage space by not having to copy data into regular tables

Users who want to use just the IWA for quick analytics on data that are available externally, in a flat file or otherwise, can now avoid the overhead of loading the data first to Informix before the data could be used for any analytics at IWA speed. Complex analytics can also be performed by doing joins between various tables that can be setup among external flat files.

Appendix A: **NoSQL Background Information**

Different Philosophies

NoSQL systems have become popular for their ability to solve two categories of problems: reducing development time and lowering query response time under very heavy load conditions. The most significant difference to working with NoSQL is that in the relational paradigm, the definitions of objects exist within the database, but in the NoSQL domain, the definitions exist and are maintained in the application. In a NoSQL paradigm, the application(s) define the structures, and the database is simply a means to store the data as serialized structures.

Where traditional relational systems provide the infrastructure behind systems of record, NoSQL systems provide the infrastructure behind systems of engagement. This is perhaps painting with too broad a brush, but it provides a broader context for how the Informix NoSQL features enhance your ability to deliver solutions to your customers. The lack of fixed structures at the database level fits developers' need to rapidly change software that interacts, or engages, with customers, and web or mobile systems of engagement tend to be more dynamic than the more mature systems of record.

Which approach is better, relational or NoSQL, depends on various aspects of the data and the nature of the applications using it. For example, some collections of data do not require interactions with other collections of data; so, there is little justification for maintaining relational data definitions within the database, separate from the application using the data. Splitting data into shards adds complexity to the infrastructure and adds a network and processing latency to query response time, but allows capacity to be added with low-cost hardware.

Application Development

JavaScript Object Notation (JSON) is the most common format used for exchanging data between the database and the application. It is a very simple interchange format for encoding, or serializing, structured objects as text. It is natively supported in JavaScript and Python, and there are readily available libraries supporting it in Java, PHP, C++, and others. (JavaScript is an entirely different language from Java, despite the name similarity.) See *DB2 NoSQL JSON capabilities, Part 1* in the ***IBM References*** for more information on the advantages of using JSON.

The differences in philosophy are reflected in the way the database systems deal with other functionality related to database management, such as incorporating business logic into the database. When you have a well-defined schema in the database layer, it is possible to define such things as constraints between values across entity types at the lowest, most common layer. Since there are no defined schemas at the database layer within a NoSQL store, it is not possible to create well-defined relationships at that level, and without those relationships, it is not practical to create a constraint within the database level that says something to the effect of, "Do not accept new rows (documents) where the value of state does not exist within the list of states in the country." So, these kinds of business logic rules have to be implemented in an application layer above the database.

Relational database systems are built to handle multi-statement transactions. Achieving the same all-or-nothing behavior in a NoSQL system like MongoDB, which does not implement the concept of transaction states, requires considerably more application code and effort to guarantee that multiple operations are execute as atomic units.

If you have an application with a short lifetime, it might make sense to spend less time on development. Application development time is reduced by enabling the application to define the structure of objects rather than keeping the structure definitions in a separate system, the database server. If an application developer wants to add, change the type, or remove a field, they only have to change the application code and do not have to coordinate their work with a database administrator, who might then have to propagate the change to shards or replicates of the data at different sites.

The ability of a developer to access information in a database without having to write SQL code, especially if that information is not inherently relational, is desirable for a large percentage of web application developers. The SQL language is primarily for expressing relationships between sets of data. If there are few, or only very simple, relationships between the sets the application uses, there is less reason for the developer to have to know SQL in order to write their application code.

When there is a requirement for more than one application to access the same data, it is useful to have some form of master for the schema. Although you may be able to create one master application that defines the structure of objects for all applications, it is advantageous to define the schema where the data are stored. This is also useful in situations where a single application changes over time, either as a result of an evolution of versions or because it is replaced by another application. Let's say that there are N applications which access the database. If the structure of the data is maintained at the database, there are N relationships between data definitions and applications that need to be maintained. The graph of relationships is a simple tree with the database at the root and every application is a branch. If there is no common definition of the data, the graph becomes more filled in, with potentially N(N+1)/2 relationships that need to be maintained. It is possible to define a master application that sets the standard for all others, but it would be easier to enforce consistent definitions if they are kept with the data.

Customer information is an example of information that is shared across multiple applications. The customer data of a company have a set of common fields that change little over time. This type of data might be maintained for the entire life of the company. However, the applications used to access the data are likely to change. It is also likely that the company has a limited number of employees who require access to such customer information, and so there is less need for horizontal scale-out. Relational database systems were specifically designed for this kind of work, maintaining customer transaction records, and they still excel at it.

Runtime Performance

NoSQL systems commonly increase throughput by relaxing the rules for atomicity, consistency, isolation, and durability (ACID), or sharding data across separate physical devices. For example, in the MongoDB default usage, the server does not wait for data to be written to disk before it acknowledges the receipt. It then returns a successful operation message to the application. Thereby, write performance can be improved at the cost of losing the guarantee that the changes are made durable. In a NoSQL sharding setup, query response time is reduced by distributing the work of the query to separate systems, each of which contains a shard, a distinct subset of a collection.

Informix maintains the durability aspect of database ACID requirements and so its behavior is roughly comparable with the MongoDB write concern level *journaled*. When write assurance is required, performance is reduced by waiting for the data to be made persistent before returning a status message to the client. Applications that can tolerate rare occurrences of data loss can increase the number of write operations by not waiting for data persistence. Write failures are rare whether they are the result of some failure of the storage media (disk drive), network packet loss, or other causes, and not all pieces of information are critical. For example, it may be that losing 1 out of 100,000 (five '9's reliability) messages in a social media hub is an acceptable tradeoff for the ability to handle a higher volume of messages.

NoSQL systems achieve schema-less databases by storing the metadata with every instance of the data. This requires more storage space than storing a single copy of the metadata for many instances of data. The amount of extra space required by a NoSQL system is greater when there are more rows, and it is greater when the metadata, like the name of a field, is relatively large. A two-character name of a 256-character document field requires about one percent more storage space than a relational column. An eight-character field name on a field that contains an average of eight characters doubles the storage space requirements. A relational DBMS can have advantages when storage space and disk I/O are important factors.

Master-detail relationships, which are trivial in a relational system, force a developer to make a choice in a NoSQL system between embedding and linking. Embedding the details in the same document as the master enables the master and details to remain atomic and consistent with each other, but can result in duplicated data. Aside from the inefficient use of storage space, any time there are duplicates of what should be the same information, extra work has to be performed to maintain consistency. Embedding details within a single document also increases the risk that the maximum size of a document will be reached. Embedding can have a performance advantage because all the information is stored contiguously on disk; so, reads and writes are efficient. Linking removes the problems of redundancy and possible size restrictions, but introduces the need to perform multiple database operations to access the same information.

Summary

You will have to decide on various tradeoffs to make between using a relational model and a NoSQL model when designing your software solution. Both the areas of application development and runtime performance present options you will want to consider. IBM Informix offers you one database system capable of supporting your solution no matter which set of options work best for you.

Appendix B: **NoSQL Additional Reading**

The following list of links provides useful background information for new NoSQL users. These provide information about how non-relational database technology is being used and how these capabilities are introduced into the IBM DB2 and Informix DBMS. The inclusion of 3rd-party material is not an endorsement of the material by IBM, but is meant to offer the reader a variety of information and viewpoints.

IBM References

DB2 NoSQL for JSON Capabilities Part IV: Wire Listener
DB2 NoSQL JSON enables developers to write applications using a popular JSON-oriented query language created by MongoDB to interact with data stored in IBM DB2 for Linux, UNIX, and Windows.
http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson4/index.html
This article describes the listener as used to access DB2; however, the Informix implementation is very similar. There are differences in installation and configuration of the listener, but the usage and examples provided are useful in understanding the Informix implementation. There are links to MongoDB resources.

DB2 NoSQL JSON capabilities, Part 1: Introduction to DB2 NoSQL JSON
Rapidly changing application environments require a flexible mechanism to store and communicate data between different application tiers. JSON (Java™ Script Object Notation) has proven to be a key technology for mobile, interactive applications by reducing overhead for schema designs and eliminating the need for data transformations.
http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson1/index.html

An introduction to MongoDB
http://www.ibm.com/developerworks/offers/lp/demos/summary/j-jmongodb.html

Explore MongoDB
Learn why this database management system is so popular
http://www.ibm.com/developerworks/library/os-mongodb4/

When NoSQL makes better sense than MySQL (The Tech Trek) – IBM
https://www.ibm.com/developerworks/community/blogs/theTechTrek/entry/when_nosql_makes_better_sense_than_mysql8

IOD 2011: Curt Cotner on NoSQL, Hadoop, Capture Replay Technology
http://www.ibm.com/developerworks/podcast/iod2011-curtcotner

Non-IBM References
How NoSQL and Relational Database Storage Can Coexist
Their differences complement one another, with each delivering functionality that the other cannot. They occupy different but equally important pieces of the database pie.

http://www.devx.com/dbzone/Article/45636

Transitioning Relational to NoSQL
How to change the way you think about data modeling
http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Couchbase_Whitepaper_Transitioning_Relational_to_NoSQL.pdf

MongoDB
MongoDB (from "humongous") is an open source document-oriented database system developed and supported by MongoDB, Inc.
http://en.wikipedia.org/wiki/MongoDB

Shard (database architecture)
A database shard is a horizontal partition in a database or search engine. Each individual partition is referred to as a shard or database shard.
http://en.wikipedia.org/wiki/Shard_(database_architecture)

Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan
Traditional enterprise applications may extend over 15 years. Enterprise mobile apps on the other hand are rewritten in a mere 14 months. This means enterprises should plan for the future of their mobile applications, but develop them for today.
http://thinkmobile.appcelerator.com/blog/bid/249258/Nothing-is-Certain-Except-Death-Taxes-and-a-Short-Mobile-App-Lifespan

When it's still best to use a relational DBMS
For short-request processing, both document stores and fully object-oriented DBMS can make sense.
http://www.dbms2.com/2011/05/29/when-to-use-relational-database-management-system/

Database management system choices – beyond relational
This is the fifth of a five-part series on database management system choices.
http://www.dbms2.com/2008/02/15/non-relational-database-management

Systems of Engagement and the Future of Enterprise IT: A Sea Change in Enterprise IT
http://www.aiim.org/futurehistory

How to Integrate Your Systems of Engagement into Your Systems of Record
ECM & Social Business Community Blog
http://www.aiim.org/community/blogs/expert/How-to-Integrate-Your-Systems-of-Engagement-into-Your-Systems-of-Record

Open Standards
JSON
JSON (JavaScript Object Notation) is a lightweight data-interchange format.
http://www.json.org/

BSON Specification
BSON - Binary JSON
http://bsonspec.org/#/specification

Appendix C: **References**

- IBM Informix database server 12.10.xC1: A Technical White Paper
- Release Notes for IBM Informix 12.10.xC2
- IBM Informix 12.10 .NET Provider Reference Guide
- IBM Informix 12.10 Administrator's Reference
- IBM Informix 12.10 Backup and Restore Guide
- IBM Informix 12.10 Database Extensions User's Guide
- IBM Informix 12.10 Enterprise Replication
- IBM Informix 12.10 GLS User's Guide
- IBM Informix 12.10 Guide to SQL: Reference
- IBM Informix 12.10 Guide to SQL: Syntax
- IBM Informix 12.10 Migrating and upgrading
- IBM Informix 12.10 Performance Guide
- IBM Informix 12.10 Security
- IBM Informix 12.10 TimeSeries Data User's Guide
- IBM Informix 12.10 Warehouse Accelerator Administration Guide

For more information

To learn more about the Informix features, contact your IBM representative or IBM Business Partner, or visit ibm.com/software/data/informix

IBM Informix Introducing NoSQL Capabilities A Technical White Paper September 2013 Chris Golledge Anup Nair © Copyright 2013 IBM Corporation IBM Corporation Software Group Route 100 Somers, NY 10589 U.S.A. *The information contained in this publication is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this publication, it is provided AS IS without warranty of any kind, express or implied. In addition, this information is based on IBM's current product plans and strategy, which are subject to change by IBM without notice.*	*IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this publication or any other materials. Nothing contained in this publication is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software.* *References in this publication to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in this presentation may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. Nothing contained in these materials is intended to, nor shall have the effect of, stating or implying that any activities undertaken by you will result in any specific sales, revenue growth, savings or other results.* *IBM, the IBM logo, ibm.com, and Informix are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.*