Informix NoSQL-SQL-Crossover

Mongo, Json, REST, and your existing data

Sprecher
andreas.legner@de.ibm.com

Agenda

- Informix as a Json Document Store
- NoSQL extending SQL
- SQL and other Informix technologies for NoSQL
- Crossbreeding within DB server
- Crossbreeding within Json/REST Listener

Informix as mongoDB **document store** … and more

Client speaking ‚*mongoDB*' can use **Informix as Mongo backend**

- Create/Drop database
- Create/Drop Table
- Store / Retrieve / Update Data
- Index Data
- **Full Informix logging, locking and transaction support**
- **Fully able to use all kinds of replication, BaR, cloning, …**
- **Able to use transparent Data Sharding (Distribution across many Nodes)**
- More …

The database used is a regular Informix DB

- Could even be a pre-existing DB
- Document Collections would be ‚**collection tables**'
- Standard SQL can access them

JSON Document (pretty() printed)

```
{
        "customer_num" : 104,
        "name" : {
                "f_name" : "Anthony",
                "l_name" : "Higgins"
        },
        "address" : {
                "address1" : "422 Bay Road"
                "city" : "Redwood City",
                "state" : "CA",
                "zipcode" : 94026
        },
        "order_dates" : [
                {
                        "order_num" : 1001,
                        "order_date" : ISODate("2008-05-19T22:00:00Z"),
                        "ship_date" : ISODate("2008-05-31T22:00:00Z")
                        "more_details" : "..."
                },
                {
                        "order_num" : 1003,
                        "order_date" : ISODate("2008-05-21T22:00:00Z"),
                        "ship_date" : ISODate("2008-05-22T22:00:00Z")
                        „other_details" : "..."                 },
            ... more orders ...
        ]
}
```

Key : Value Pair

Sub-Doc

Array Sub-Doc

NoSQL data types: JSON & BSON

New Informix data types:

- **JSON**:
  - Java Script Object Notation
  - UTF **string representation**
  - Made up of key-value pairs, (optionally) nested sub-documents
  - Syntax: Mongodb Standard
  - **Display type** – the one to use in e.g. dbaccess queris

- **BSON**
  - Binary
  - more space efficient representation of JSON
  - For **storage** and **transport** (database <-> client)
  - Table column type – the one to use for CREATE TABLE
  - Many functions available to act on document member elements

Caveate - Considerations

- JSON / BSON assuming/using **UTF** !
  - Only UTF8 within JSON data
  - Wire listener (JDBC) code set converting non-JSON data
  - genbson() UDR has problems with non-UTF8 data
  - Best pratice: **UTF8 database!**

- Mongo/REST API User **Authentication**
  - 2 levels:
    - Listener connecting to db: JDBC using Informix auth persistent connection for all mongo requests
    - Mongo API having its own auth
  - maintaining own auth tables in database
  - Changing with mongo version
  - Table level privileges?
  - Haven‘t much dealt with yet …

NoSQL: Schema Free !

- JSON docs have no fixed schema
  - Can have as **many/few fields** as they want
  - **Same field‘s value** in different docs can have **different types**
    - Would still go into same index

- Any doc could go into any collection
  - Could have docs describing all kinds of things in one collection
  - (You‘d still keep things separately, if only for performance)

- Docs describing same kind of things can have different attributes
  - E.g. jpeg pictures from different camera types

Schema Free – Opening up new Horizons

Imagine: BSON column instead of many SQL columns:

- Different applications using/maintaining different sets of attibutes
- No longer need to agree and work all on same schema
- Apps could change without need for DB schema change
  - No more ALTER TABLE … ?

BSON in TimeSeries Row type:

- Rather than fixed schema row type
- Allowing great variety of IoT data sources / devices
- Allowing new types of IoT devices easily

Informix SQL <-> mongoDB NoSQL

DB Server:

- JSON/BSON NoSQL data **available to SQL** in all detail
  - Not only visible to / usable by mongo clients
  - Join to relational data
  - Utilise BSON indices
  - Insert/Delete/Update through SQL
  - Define Views incl. both NoSQL and relational data
  - Accessible by SPL/Triggers
  - TBD: accessible via MI API (direct access in C UDRs)

- Relational data in JSON/BSON representation
  - Through Casting
  - Using **`genbson()`** UDR

Informix SQL <-> mongoDB NoSQL

JSON / REST Wire Listener:

- Mongo API can access
  - true JSON doc collections (created through this API)
  - other tables containing BSON columns
  - pure relational tables (no joins – not supported by API)
  - views (so allowing joins this way)

- Any of these addressable
  - as ‚collections': wire listener acting as translator
  - using ‚passthrough' SQL:
    any SQL possible from mongo/REST client

- Returned data (and any return values) in JSON format

Informix „Mongo“ API: jsonListener

JDBC based translation agent providing Mongo API

Mongo queries:

- **`db.<collection_name>.find/insert/update/delete(…)`**
- Also working on traditional SQL tables
  - <collection_name>: your table‘s (or view‘s) name

Passthrough SQL (security.sql.passthrough=true):

- Special <collection_name>: **`system.sql`**
- db.system.sql.find({$sql: „**`<any_possible_SQL>`**“})

REST Listener/API

- Different run mode of JSON Listener
- Internally: a REST layer ontop of Mongo API
- Hence providing much the same functionality

Basic query syntax:

- `http://<server>:<port>/db/collection`
  `?query={..}&fields={...}&batchsize=<#>`
- Also fully supporting system.sql

JSON/BSON functions – SQL looking into JSON docs

- **`bson_get(BSON/JSON, „key“)`**
  - return a key:value pair (BSON), can be complex / sub-doc

- **`bson_value_XXX(BSON, „key“)`**
  - Return value of a key in Informix type
  - XXX: most important Informix types
  - `integer, bigint, double` (-> FLOAT)
  - `date, timestamp` (both -> DATETIME)
  - `boolean, lvarchar, varchar`

- Less known:
  - bson_concat(), bson_create(), bson_rename(), bson_new() bson_keys_exist(), …

Accessing JSON Doc Elementes: **Dot Notation**

Alternate way for accessing (reading!) a **key:value** pair or **value**

- **bsondoc.path.to.key**
  - data.address.city
- Returns bson (key:value pair, maybe complex)
- Can be cast to Informix type – if leaf key

- ```
  select data.address::json from test_coll
   where data.address.city::varchar = "Berlin"
  ```
  ```

Anything Goes …

All these fully valid SQL expressions

-> usable in

- Projection clause
- WHERE clause
- ORDER BY clause
- VIEWs
- Functional Indices (BSON Indices)
- Stored Procedures
- Triggers
- Cast definitions
- Distributed queries (across databases/servers)

BSON Updates

- **`bson_update(BSON, „update-doc“)`**
  - Modify a BSON doc
  - at key/sub-key level
- Supported UPDATE operators:
  - $set update key‘s value
  - $unset delete key
  - $put adds a key:value pair
  - $min/$max
  - $inc/$mul increment/multiply a value
  - $rename … a key
  - $replace … a key
  - …

genBSON() – Relational Data to Document Store

- **genBSON(<ROW_type>, keep_nulls, skip_id)**
  - Transforms row/tuple data to BSON doc
  - Optionally keep field with NULL values
  - Optionally suppress JSON ObjectId generation
  - not all data types supported & needs UTF8 input

Transfer a regular table to a JSON collection:

- `create collection table mycoll ( data BSON);`
- ```
  insert into mycoll (data)
    select genbson (customer) from customer;
  ```

or

- ```
  insert into mycoll (data)
    select genbson (ROW(field_list))
      from some_complex_view;
  ```

Fragen?

Sprecher
andreas.legner@de.ibm.com