Introduction to Data Structure (Data Management) Lecture 11

Felipe P. Vista IV

Reminder

- Everybody, make sure that your name in ZOOM is in the following format:
  - University ID Num Name (no “( )“)
  - Ex: 202054321 Juan Dela Cruz
  -
  - Not changing your name to this format
    - you might be marked Absent
    - * → absent?

- JSON (Continuation)
- AsterixDB
- SQL++

INTRO TO DATA STRUCTURE

JSON (CONTINUATION)

JSon Data

- JSon is self-describing

JSon Data

- JSon is self-describing
- Schema elements become part of the data
  - Relational schema: person(name, phone)
  - In Json: “person”, “name”, “phone” are part of the data, are are repeated many times

JSon Data

- JSon is self-describing
- Schema elements become part of the data
  - Relational schema: person(name, phone)
  - In Json: “person”, “name”, “phone” are part of the data, are are repeated many times
- Consequence: JSon is much more flexible
  - also uses more space (but can be compressed)

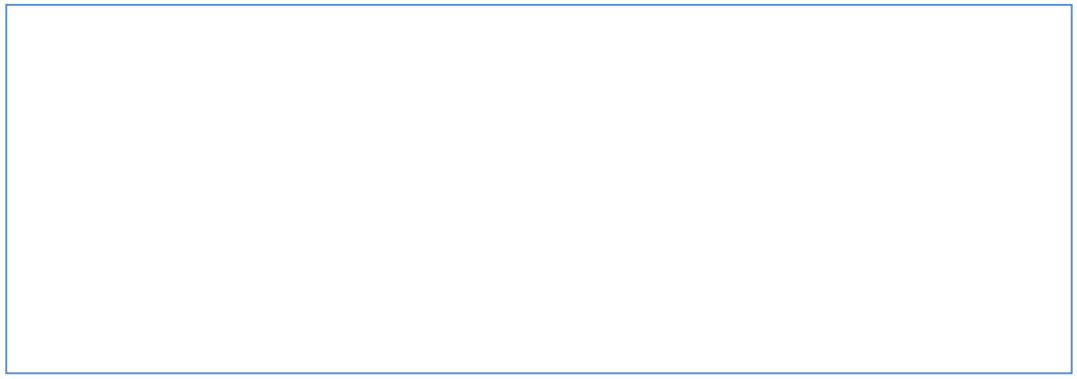
JSon Data

- JSon is self-describing
- Schema elements become part of the data
  - Relational schema: person(name, phone)
  - In Json: "person", "name", "phone" are part of the data, are are repeated many times
- Consequence: JSon is much more flexible
  - also uses more space (but can be compressed)
- JSon is an example of semi-structured data

Mapping Relational Data to JSON

Person

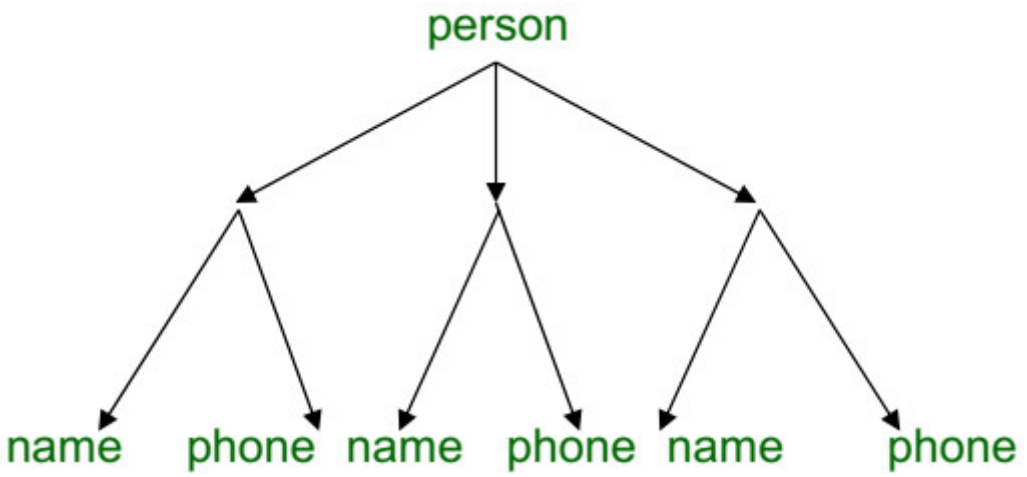
name	phone
Janin	1234
Mikki	5678
Soheil	9012



Mapping Relational Data to JSON

Person

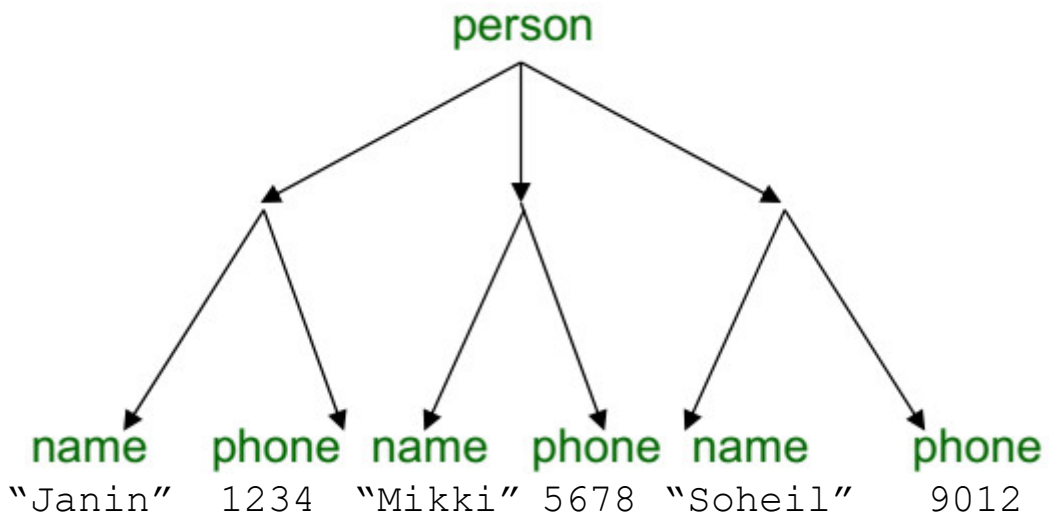
name	phone
Janin	1234
Mikki	5678
Soheil	9012



Mapping Relational Data to JSON

Person

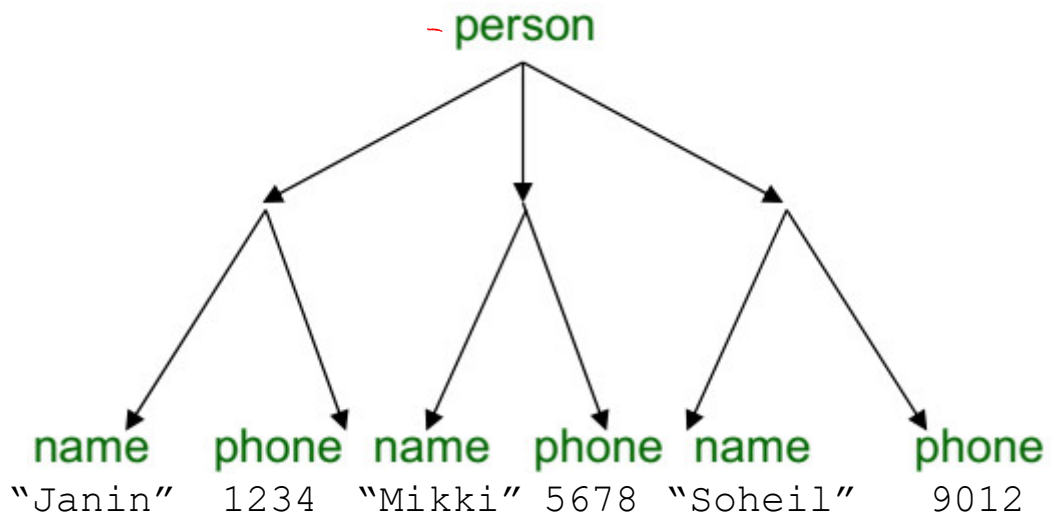
name	phone
Janin	1234
Mikki	5678
Soheil	9012



Mapping Relational Data to JSON

Person

name	phone
Janin	1234
Mikki	5678
Soheil	9012




```
{"person":
    [{"name":"Janin", "phone":1234},
     {"name":"Mikki", "phone":5678},
     {"name":"Soheil", "phone":9012}
    ]
}
```

Mapping Relational Data to JSON

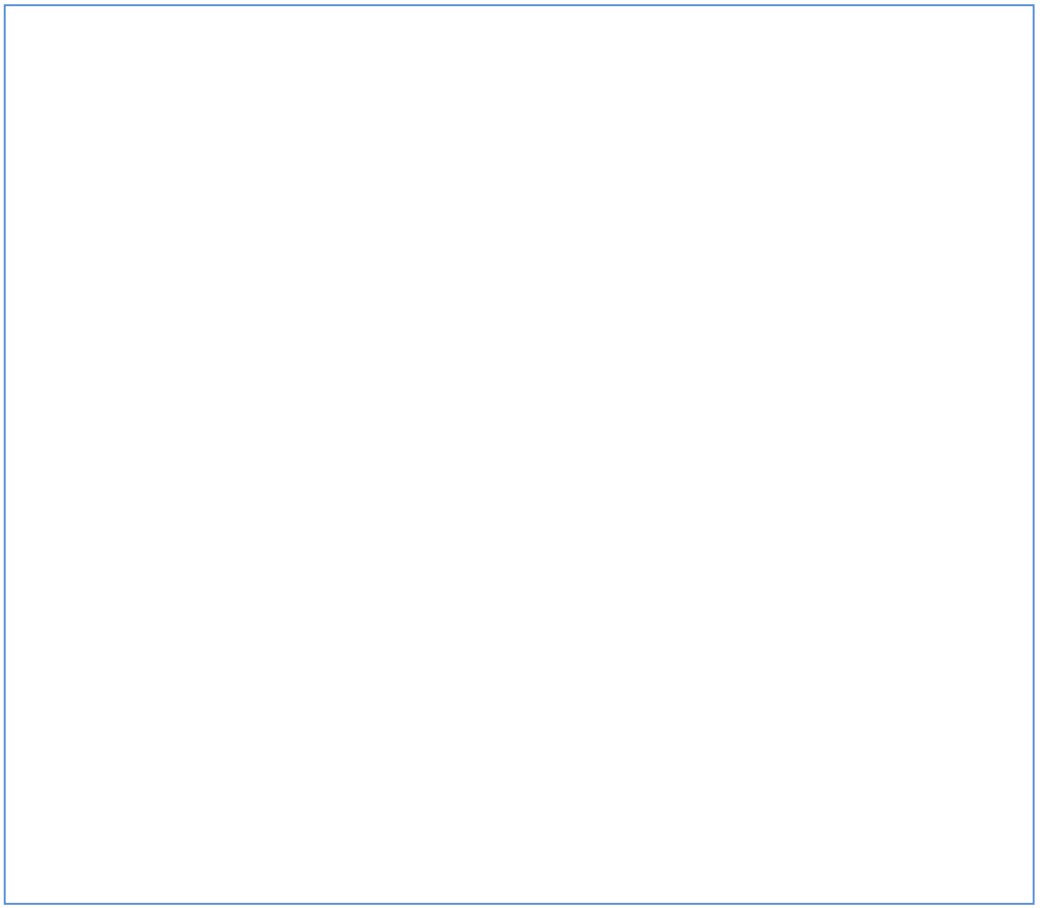
Inline Foreign Keys

Person

name	Phone
Janin	1234
Mikki	5678

Orders

personName	date	product
Janin	2012	Bike
Janin	2014	Scooter
Mikki	2012	Scooter



Mapping Relational Data to JSON

Inline Foreign Keys

Person

name	Phone
Janin	1234
Mikki	5678

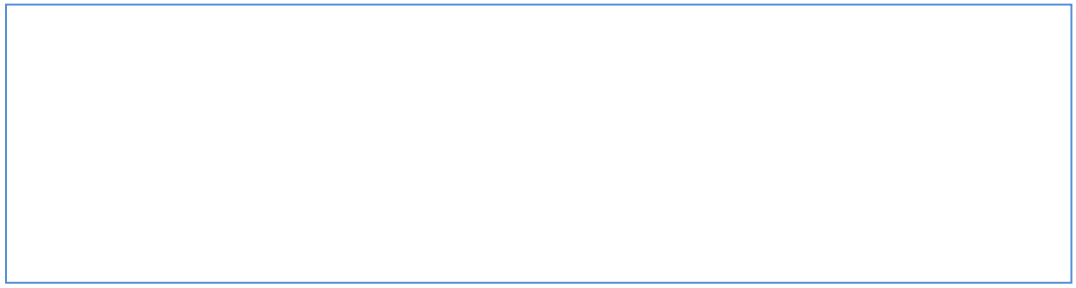
Orders

personName	date	product
Janin	2012	Bike
Janin	2014	Scooter
Mikki	2012	Scooter

```
{"person":
    [{"name":"Janin",
      "phone":1234,
      "Orders":[{"date":2012,
                 "product":"Bike"},
                {"date":2014,
                 "product": "Scooter"}
                ]
     },
     {"name":"Mikki",
      "phone":5678,
      "Orders":[{"date":2002,
                 "product": "Scooter"}
                ]
     }
    ]
}
```

JSON = Semi-structured Data (1/3)

- Missing attributes



JSON = Semi-structured Data (1/3)

- Missing attributes

```
{"person":
   [{"name":"Janin", "phone":1234},
    {"name":"Pat"}
   ]
}
```

No phone!

JSON = Semi-structured Data (1/3)

- Missing attributes

```
{"person":
   [{"name":"Janin", "phone":1234},
    {"name":"Pat"}
   ]
}
```

No phone!

- Could represent a table with nulls

name	Phone
Janin	1234
Pat	(-)

JSON = Semi-structured Data (2/3)

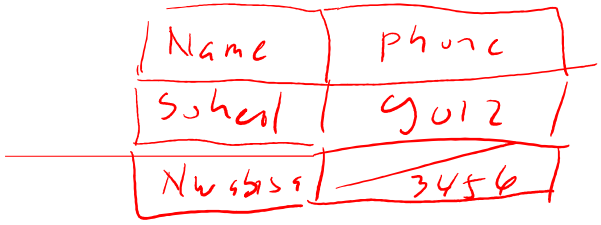
- Repeated attributes

JSON = Semi-structured Data (2/3)

- Repeated attributes

```
{"person":
   [{"name":"Soheil", "phone":9012},
    {"name":"Nwabisa", "phone":[3456,7890]}]
}
```

Two phones!



JSON = Semi-structured Data (2/3)

- Repeated attributes

```
{"person":
   [{"name":"Soheil", "phone":9012},
    {"name":"Nwabisa", "phone":[3456,7890]}]
}
```

Two phones!

- Impossible in one table

name	Phone		
Nwabisa	3456	7890	???
Nwabisa	7890		

JSON = Semi-structured Data (3/3)

- Attributes w/ different types in different objects

JSON = Semi-structured Data (3/3)

- Attributes w/ different types in different objects

```
{"person":
  [{"name":"Divan", "phone":4321},
   {"name":{"first":"Jenny", "last":"Rhee"}, "phone":8765}]
}
```

Structured name!

JSON = Semi-structured Data (3/3)

- Attributes w/ different types in different objects

```
{"person":
   [{"name":"Divan", "phone":4321},
    {"name":{"first":"Jenny"}, "last":"Rhee" "phone":8765}]
}
```

Structured name!

- Nested collections
- Heterogenous collections

Discussion

- Data exchange formats
  - well suited for exchanging data between apps
  - XML, JSON, Protobuf

Discussion

- Data exchange formats
  - well suited for exchanging data between apps
  - XML, JSON, Protobuf

- Increasingly, some systems use them as a data model
  - SQL Server: support for XML-valued relations
  - CouchBase, MongoDB: JSon as data model
  - Dremel (BigQuery): Protobuf as data model

Query Languages for Semi-Structured Data

- XML: XPath, XQuery
  - Supported by many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines

Query Languages for Semi-Structured Data

- XML: XPath, XQuery
  - Supported by many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf
  - Used internally by Google, externally in BigQuery
  - Similar to compiled JSON

Query Languages for Semi-Structured Data

- XML: XPath, XQuery
  - Supported by many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf
  - Used internally by Google, externally in BigQuery
  - Similar to compiled JSON
- JSON
  - CouchBase:N1QL; MongoDB: has pattern-based language
  - JSONiq (http://www.jsoniq.org)
  - AsterixDB: SQL and SQL++

INTRO TO DATA STRUCTURE

AsterixDB

AsterixDB

- NoSQL database system (document store)
- Develop at UC Irvine
  - Now an Apache project
- Designed to be installed on a cluster
  - Multiple machines (nodes) together implement the DBMS
  - Allows scaling to much larger amounts of data
- Weak support for multi-node transactions
- Good support for multi-node queries

AsterixDB (con't.)

- Data is partitioned over nodes by Primary Key
  - Queries involve not only disk but also Network I/O

AsterixDB (con't.)

- Data is partitioned over nodes by Primary Key
  - Queries involve not only disk but also Network I/O

- Support advanced queries
  - joins, nested queries, grouping & aggregation

AsterixDB (con't.)

- Data is partitioned over nodes by Primary Key
  - Queries involve not only disk but also Network I/O
- Support advanced queries
  - joins, nested queries, grouping & aggregation
- No statistics maintained yet (per docs)
  - May need more hints to get good performance
  - Expected this aspect to improve

AsterixDB and SQL++

- Asterix's own language is SQL
  - Based on XQuery (for XML)

AsterixDB and SQL++

- Asterix's own language is SQL
  - Based on XQuery (for XML)

- SQL++
  - SQL-like syntax for AQL
  - More familiar to database users

Asterix Data Model (ADM)

- ADM is an extension of JSON

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {“Matt”:“”, “age”:30}
  - Fields must be distinct: {“Matt”:“”, “age”:30, “age”:40}

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {“Matt”:“”, “age”:30}
  - Fields must be distinct: {“Matt”:“”, “age”:30, “age”:40}
- Arrays:
  - [1, 3, “Khan”, 5, 7]
    - It can be heterogenous

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {“Matt”:“”, “age”:30}
  - Fields must be distinct: {“Matt”:“”, “age”:30, “age”:40}
- Arrays:
  - [1, 3, “Khan”, 5, 7]
    - It can be heterogenous
- Bags
  - {{1, 3, “Khan”, “Khan”, 5, 7}}

Examples

Try these queries :

```
SELECT age FROM [{'name':'Duke',
                 'age':['30','45']}] x;
```

```
SELECT age FROM {{{'name':'Duke',
                   'age':['30','45']}}} x;
```

```
-- error
SELECT age FROM {'name':'Duke',
                 'age':['30','45']} x;
```

Data Types

- Supports SQL/JSON data type:
  - boolean, integer, float (various precisions), null
- Some SQL types not in JSON
  - date, time, interval
- Some new types
  - geometry (point, line, …)
  - UUID = universally unique identifier (systems generated, globally unique key)

Null vs. Missing

- {"age": null} = the value NULL (like in SQL)
- {"age": missing} = {} = really missing
  - boolean, integer, float (various precisions), null

```
SELECT x.b FROM [{'a':1,'b':2},{'a':3}] x;
```

```
{'b':2}
{}
```

```
SELECT x.b FROM [{'a':1,'b':2},
                 {'a':3,'b':missing}] x;
```

```
{'b':2}
{}
```

INTRO TO DATA STRUCTURE

SQL++

SQL++ Overview

- Data definition language:
  - Dataverse (= database)
  - Dataset (= table)
    - each row uses a declared Type
  - Types
    - declares the required parts
    - can allow for extra data (open vs closed types)
  - Indexes
- Query language: select-from-where

Dataverse

A Dataverse is a Database

- CREATE DATAVERSE lec12
- CREATE DATEVERSE lec12 IF NOT EXISTS

- DROP DATAVERSE lec12
- DROP DATAVERSE lec12 IF EXISTS

- USE lec12

Type

- Defines the schema of a collection
- It lists all *required* fields
- Fields followed by "?" Are *optional*
- CLOSED type = no other fields are allowed
- OPEN type = other fields are allowed

Closed Types

```
USE lec12;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name: string, age: int, email: string? }
```

```
{"Name":"Perry", "age":40, "email":"j@perry.com"}
```

```
{"Name":"Paul", "age":30}
```

```
--not OK:
{"Name":"Della", "age":35, "phone": "9876543210"}
```

Open Types

```
USE lec12;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
   Name: string, age: int, email: string? }
```

```
{"Name":"Perry", "age":40, "email":"j@perry.com"}
```

```
{"Name":"Paul", "age":30}
```

```
--now it's OK:
{"Name":"Della", "age":35, "phone": "9876543210"}
```

Types with Nested Collections

```
USE lec12;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name: string, phone: string? }
```

```
{"Name":"Nwabisa", "phone":["2468"]}
{"Name":"Pat", "phone":["1357","9024"]}
{"Name":"Evan", "phone":[]}
```

Datasets

- Dataset = relation
- Must have a type
  - can be a trivial OPEN type
- Must have a key
  - can be declared "autogenerated" if UUID
  - (SQL systems usually support auto-incremented unique integer IDs)

Dataset with Existing Key

```
USE lec12;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name: string, email: string? }
```

```
{"Name":"Khan"}
{"Name":"Matt"}
…
```

```
USE lec12;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
     PRIMARY KEY Name;
```

Dataset with Auto Generated Key

```
USE lec12;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  myKey: uuid, Name: string,
  email: string? }
```

```
{"Name":"Khan"}
{"Name":"Matt"}
…
```

Note: no **myKey** value since it will be auto-generated

```
USE lec12;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
     PRIMARY KEY myKey AUTOGENERATED;
```

Discussion of NFNF

- NFNF = Non First Normal Form
  - one or more attributes contain a collection
- One extreme:
  - a single row with a huge nested collection
- Better:
  - multiple rows, reduced number of nested collections

Example

mondial.adm is totally semi-structured:

- {“mondial”:{“country”:[…], “continent”:[…], …, “desert”:[…]}}

country	continent	organization	sea	…	mountain	desert
[{“name”: “Albania”, …}, {“name”: “Greece”, …}, …]	…	…	…		…	…

- country.adm, sea.adm, mountain.adm are more structured

-car_code	name	…	ethnicgroups	religions	…	city
AL	Albania	…	[…]	[…]	…	[…]
GR	Greece	…	[…]	[…]	…	[…]
…	…	…	…			

Indexes

- Can declare an index on an attribute of a top-most collection
- Available:
  - BTREE: good for equality and range queries

    E.g. name="Greece"; 20 < age and age < 40
  - RTREE: good for 2-dimensional range queries

    E.g. 20 < x and x < 40 and 10 < y and y < 50
  - KEYWORD: good for substring search

Indexes

Cannot index inside a nested collection

```
USE lec12;
CREATE INDEX countryID
    ON country(name)
    TYPE BTREE;
```

```
USE lec12;
CREATE INDEX cityname
    ON country(city.name)
    TYPE BTREE;
```

Country:

-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...	...	...	...			
BG	Belgium	...				
...						

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {“Matt”:“”, “age”:30}
  - Fields must be distinct: {“Matt”:“”, “age”:30, “age”:40}
- Arrays:
  - [1, 3, “Khan”, 5, 7]
    - It can be heterogenous
- Bags
  - {{1, 3, “Khan”, “Khan”, 5, 7}}

Examples

Try these queries :

```
SELECT age FROM [{'name':'Duke',
                  'age':['30','45']}] x;
```

~>{"age": ["30", "50"]}

```
SELECT age FROM {{{'name':'Duke',
                   'age':['30','45']}}} x;
```

~>{"age": ["30", "50"]}

```
--error
SELECT age FROM {'name':'Duke',
                 'age':['30','45']} x;
```

SQL++ Overview

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```

Retrieve Everything

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

```
SELECT x.mondial FROM world x;
```

Answer:

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  …
  …
}
```

Retrieve Countries

```
{"mondial":
 {"country":[country1, country2, …],
   "continent":[…],
   "organization":[…],
   ...
   ...
 }
```

```
SELECT x.Mondial.country FROM world x;
```

Answer:

```
{"country":[country1, country2, …]}
```

Retrieve Countries, one by one

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

```
SELECT y AS country
FROM world x, x.Mondial.country y;
```

Answer:

```
{"country":country1}
{"country":country2}
...
```

Escape characters

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

Illegal field: "-car_code",
Use: '-car_code'

```
SELECT y.'-car_code' AS code, y.name AS name
FROM world x, x.Mondial.country y ORDER BY y.name;
```

Answer:

```
{"code":"AFG", "name":"Afghanistan"}
{"code":"AL", "name":"Albania"}
...
```

Nested Collections

- If the value of attribute B is some other collection, then we can simple iterate over it

```
SELECT x.A, y.C, y.D
FROM mydata x, x.B y;
```

x.B is a collection

```
{"A":"a1","B":[{"C":"c1","D":"d1"},
               {"C":"c2","D":"d2"}]}
{"A":"a2","B":[{"C":"c3","D":"d3"}]}
{"A":"a3","B":[{"C":"c4","D":"d4"},
               {"C":"c5","D":"d5"}]}
```

```
{"A":"a1","C": "c1","D":"d1"}
{"A":"a1","C": "c2","D":"d2"}
{"A":"a2","C": "c3","D":"d3"}
{"A":"a3","C": "c4","D":"d4"}
{"A":"a3","C": "c5","D":"d5"}
```

Heterogeneous Collections

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
}
```

Runtime error!

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name='Greece';
```

The problem:

```
…
"province":[…
 {"name":"Yah",
  "city":[{"name":"Nabas"…},{"name:Buruanga"…},…] …},
 {"name":"Sya",
  "city":{"name":"Banwa"…} …},
```

city is an array

city is an object

Heterogeneous Collections

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name='Greece' AND is_array(z.city);
```

Just the arrays

The problem:

```
…
"province":[…
 {"name":"Yah",
  "city":[{"name":"Nabas"…},{"name:Buruanga"…},…] …},
 {"name":"Sya",
  "city":{"name":"Banwa"…} …},
```

Heterogeneous Collections

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

Note: get name directly from z

```
SELECT z.name AS province_name, z.city.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name='Greece' AND NOT is_array(z.city);
```

Just the objects

The problem:

```
…
"province":[…
 {"name":"Yah",
  "city":[{"name":"Nabas"…},{"name:Buruanga"…},…] …},
 {"name":"Sya",
  "city":{"name":"Banwa"…} …},
```

Heterogeneous Collections

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

if z.city is array
  u <= z.city
else
  u <= makeArray(z.city)

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z,
     [CASE WHEN is_array(z.city) THEN z.city
           ELSE [z.city] END) u
WHERE y.name='Greece';
```

u - always be an array

Get both!

The problem:

```
…
"province":[…
 {"name":"Yah",
  "city":[{"name":"Nabas"…},{"name:Buruanga"…},…] …},
 {"name":"Sya",
  "city":{"name":"Banwa"…} …},
```

Heterogeneous Collections

```
{"mondial":
 {"country":[country1, country2, …],
  "continent":[…],
  "organization":[…],
  ...
  ...
}
```

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z,
     CASE WHEN z.city is_missing THEN []
          WHEN is_array(z.city) THEN z.city
          ELSE [z.city] END) u
WHERE y.name='Greece';
```

Even better!

The problem:

```
…
"province":[…
 {"name":"Yah",
  "city":[{"name":"Nabas"…},{"name:Buruanga"…},…] …},
 {"name":"Sya",
  "city":{"name":"Banwa"…} …},
```

Reminder

- Everybody, make sure that your name in ZOOM is in the following format:
  - University ID Num Name (no “( )“)
  - Ex: 202054321 Juan Dela Cruz
  -
  - Not changing your name to this format
    - you might be marked Absent
    - * → absent?

Useful Functions

- is_array
- is_boolean
- is_number
- is_object
- is_string
- is_null
- is_missing
- is_unknown = is_null or is_missing

Basic Unnesting

- An array: [a, b, c]
- A nested array: arr = [[a, b], [], [b, c, d]]
- Unnest(arr) = [a, b, b, c ,d]

```
SELECT y
FROM arr x, x y
```

Unnesting Specific Field

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}],
 {A:a2, F:[{B:b3},{B:b4} ,{B:b5}], G:[]},
 {A:a3, F:[{B:b6}], G:[{C:c2}, {C:c3}]}]
```

```
Unnest_F(coll) =
[{A:a1, B:b1, G:[{C:c1}]},
 {A:a1, B:b2, G:[{C:c1}]},
 {A:a2, B:b3, G:[]},
 {A:a2, B:b4, G:[]},
 {A:a2, B:b5, G:[]},
 {A:a3, B:b6, G:[{C:c2}, {C:c3}]}]
```

New RA expression

```
Unnest_G(coll) =
[{A:a1, F:[{B:b1}, {B:b2}], C:c1},
 {A:a3, F:[{B:b6}], C:c2},
 {A:a3, F:[{B:b6}], C:c3}
```

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

SQL++

```
SELECT x.A, x.F, z.C
FROM coll x, x.G z
```

Nesting (like group-by)

A flat collection

```
coll =
[{A:a1,B:b1}, {A:a1,B:b2}, {A:a2,B:b1}]
```

New RA expression

```
Nest_A(coll) =
[{A:a1, GRP:[{B:b1}, {B:b2}]},
 {A:a2, GRP:[{B:b1}]}]
```

```
Nest_B(coll) =
[{B:b1, GRP:[{A:a1}, {A:a2}]},
 {B:b2, GRP:[{A:a1}]}]
```

```
SELECT DISTINCT x.A,
   (SELECT y.B FROM coll y WHERE x.A = y.A) AS GRP
FROM coll x
```

```
SELECT DISTINCT x.A AS GRP
FROM coll x
LET g = (SELECT y.B FROM coll y WHERE x.A = y.A)
```

Group-by / Aggregate

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}],
 {A:a2, F:[{B:b3},{B:b4} ,{B:b5}], G:[]},
 {A:a3, F:[{B:b6}], G:[{C:c2}, {C:c3}]}]
```

```
SELECT x.A, coll_count(x.F) AS cnt
FROM coll x
```

```
SELECT x.A, count(*) AS cnt
FROM coll x, x.F y
GROUP BY x.A
```

Count the number of elements in the **F** collection

```
coll_count() =
 collection count
```

These are not equivalent!
Why?

Group-by / Aggregate

A flat collection

```
coll =
[{A:a1,B:b1}, {A:a1,B:b2}, {A:a2,B:b1}]
```

```
SELECT DISTINCT x.A, coll_count(g) AS cnt
FROM coll x
LET g =(SELECT y.B FROM coll y WHERE x.A = y.A)
```

```
SELECT x.A, count(*) AS cnt
FROM coll x
GROUP BY x.A
```

Are these equivalent!

Join

Two flat collection

```
coll1 = [{A:a1,B:b1}, {A:a1,B:b2}, {A:a2,B:b1}]
coll2 = [{B:b1,C:c1}, {B:b1,C:c2}, {B:b3,C:c3}]
```

```
SELECT x.A, x.B, x.C
FROM coll1 x, coll2 y
WHERE x.B = y.B
```

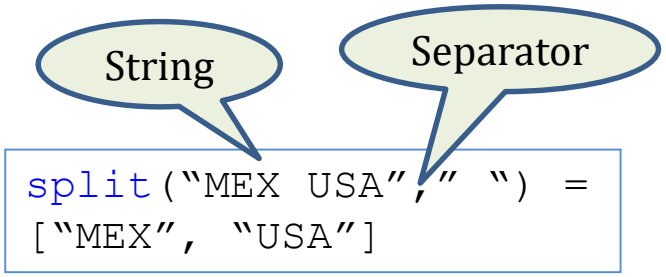
Multi-Value Join

- A many-to-one relationship should have one foreign key, from “many” to “one”
  - each of the “many” points to the same “one”
- Sometimes, people represent it in the opposite direction, from “one” to “many”:
  - Ex: list of employees of a manager
  - reference could be space-separated string of keys
  - need to use split(string, separator) to split it into a collection of foreign keys

Multi-Value Join

```
River =
[{"name":"Danube","-country":"SRB A D H HR SK BG RO MD UA"},
 {"name":"Colorado","-country":"MEX USA"},
...]
```

```
SELECT ...
FROM country x, river y,
     split(y,'-country'," ") z
WHERE x.'-car_code'= z
```



Behind the Scenes

- Query Processing on NFNF data:
  - Option 1: give up on query plans
  - Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan
- We can apply the second approach ourselves, to work with semi-structured data using a familiar RDBMS
  - For data analysis, this may be more efficient until semi-structured DBMSs catch up to RDBMSs

Flattening SQL++ Queries

A Nested Representation

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}],
 {A:a2, F:[{B:b3},{B:b4} ,{B:b5}], G:[]},
 {A:a3, F:[{B:b6}], G:[{C:c2}, {C:c3}]}]
```

Flat Representation

id	A
1	a1
2	a2
3	a1

parent	B
1	b1
1	b1
2	b3
2	b4
3	b5
3	b6

parent	C
1	c1
3	c2
3	c3

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, x.F y, x.G z
WHERE y.B = z.C
```

SQL

```
SELECT x.A, y.B
FROM coll x, F z
WHERE x.id = y.parent AND x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent AND
      x.id = z.parent AND y.B = z.C
```

Thank you.