Asterix Data Model (ADM)

- Based on the JSON standard
- Objects:
  - `{"Name": "Alice", "age": 40}`
  - Fields must be distinct:
    - `{"Name": "Alice", "age": 40, "age": 50}`
- Ordered arrays:
  - `[1, 3, "Fred", 2, 9]`
  - Can contain values of different types
- Multisets (aka bags):
  - `{1, 3, "Fred", 2, 9}`
  - Mostly internal use only but can be used as inputs
  - All multisets are converted into ordered arrays (in arbitrary order) when returned at the end

Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, ...), date, time, etc
- UUID = universally unique identifier
  Use it as a system-generated unique key

Closed Types

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

{"name": "Alice", "age": 30, "email": "a@alice.com"}
{"name": "Bob", "age": 40}
-- not OK:
{"name": "Carol", "phone": "1234567890"}
```

Open Types

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

{"name": "Alice", "age": 30, "email": "a@alice.com"}
{"name": "Bob", "age": 40}
-- now it's OK:
{"name": "Carol", "age": 20, "phone": "1234567890"}
```
Types with Nested Collections

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  Name: string,
  phone: [string]
}[
"Name": "Carol", "phone": ["1234"],
"Name": "David", "phone": ["2345", "6789"],
"Name": "Evan", "phone": []
]
```

Datasets

- Dataset = relation
- Must have a type
  - Can be a trivial OPEN type
- Must have a key
  - Can also be a trivial one

**Example from HW5**

mondial.adm is totally semi-structured:

```json
{"mondial":{"country": [...], "continent": [...], ..., "desert": [...]}}
```

Country.adm, sea.adm, mountain.adm are more structured

This is no longer 1NF

- 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

Dataset with Existing Key

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name: string,
  email: string?
}[
{"name": "Alice"},
{"name": "Bob"},...
]
USE myDB;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

Dataset with Auto Generated Key

```sql
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  myKey: uuid,
  Name: string,
  email: string?
}[
{"name": "Alice"},
{"name": "Bob"},...
]
USE myDB;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY myKey AUTOGENERATED;
```

Note: no myKey inserted as it is autogenerated
Indexes

- Can declare an index on an attribute of a top-most collection
- Available options:
  - 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 if your dataset contains strings
- Will discuss how they help later in the quarter

Indexes

- Cannot index inside a nested collection

SQL++ Overview

```sql
USE myDB;
CREATE INDEX countryCode
ON country(`car_code`) TYPE BTREE;
```

```text
Country:
- car_code
- name
- ... ethnicgroups
- religions
- ... city

AL
Albania
...

GR
Greece
...

...

BG
Belgium
...
```

Retrieve Everything

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

Answer

```json
{
"mondial":
{ "country": [{Albania}, {Greece}, ...]
,
"continent": [...] ,
"organization": [...]
,...
}
}
```

Retrieve countries

```sql
SELECT x.mondial.country FROM world AS x;
```

Answer

```json
{"country": [{Albania}, {Greece}, ...]}
```
FROM world AS x, country AS c

SELECT x.mondial.country.name, c.gdp_total
FROM world AS x, country AS c
WHERE x.mondial.country.'-car_code' = c.'-car_code';

Find each country's GDP

x.mondial.country is an array of objects. No field as '-car_code'!

Error: Type mismatch!

Need to "unnest" the array

Unnesting collections

mydata

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

Form cross product between each x and its x.B

Answer

Unnesting collections

mydata

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

Same as before
The problem:  
WHERE FROM world x, x.mondial.country y, country AS c  
SELECT y.name, c.gdp_total  
FROM world AS x, x.mondial.country AS y, country AS c  
WHERE y.name = 'Greece' AND IS_ARRAY(y.city);

Answer  
[ { "name": "Albania", "gdp_total": "4180" }  
 { "name": "Greece", "gdp_total": "101700" }  
 ... ]

CSE 414 - Autumn 2018  25

The problem:  
"name": "Greece",  
"province": [ ...  
 { "name": "Attiki",  
   "city": [ { "name": "Athens"...}, { "name": "Piraeus"...}, ... ]  
   ...  
   { "name": "Sporades",  
   "city": [ { "name": "Sporades"...}, { "name": "Ios"...}, ... ]  
   ...  
 }]  
...  
(city is an array)
Useful Functions

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

Useful Paradigms

- Unnesting
- Nesting
- Grouping and aggregate
- Joins
- Multi-value join

Nesting

We want:

```
C
{(A:a1, B:b1),
 (A:a1, B:b2),
 (A:a2, B:b2)}
```

Using LET syntax:

```
SELECT DISTINCT x.A,
   (SELECT y.B FROM C AS y WHERE x.A = y.A) AS Grp
FROM C AS x
```

Counting elements in the F array for each A

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

Using LET syntax:

```
SELECT x.A, COLL_COUNT(x.F) AS cnt
FROM C AS x
```

These are NOT equivalent!

Lesson:

Read the "$@#$ manual!!"
Joins

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) \}

Answer

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

SELECT x.A, x.B, y.C FROM coll1 AS x JOIN coll2 AS y ON x.B = y.B

Outer Joins

Two flat collection

coll1 = \{ (A:a1, B:b1), (A:a1, B:b2), (A:a2, B:b1) \}
coll2 = \{ (B:b1, C:c1), (B:b3, C:c3) \}

Answer

SELECT x.A, x.B, y.C FROM coll1 AS x RIGHT OUTER JOIN coll2 AS y ON x.B = y.B

Ordering

coll1 = \{ (A:a1, B:b1), (A:a1, B:b2), (A:a2, B:b1) \}

Answer

SELECT x.A, x.B FROM coll1 AS x ORDER BY x.A

Data type matters!

"90" > "8000" but 90 < 8000 !

Multi-Value Join

river

data: "Donau", "country": "SRB A D H HR SK BG RO MD UA"
"Colorado", "country": "MEX (USA)", ...

Answer

SELECT ...
FROM country AS x, river AS y,
WHERE x.car_code = z

split("MEX USA", ") = ["MEX", "USA"]

String Separator A collection

Behind the Scenes

Query Processing on NFN data:

- Option 1: give up on query plans, use standard java/python-like execution

- Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan

Flattening SQL++ Queries

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]))]
Semistructured Data Model

- Several file formats: JSON, protobuf, XML
- The data model is a tree
- They differ in how they handle structure:
  - Open or closed
  - Ordered or unordered
- Query language needs to take NNFNF into account
  - Various “extra” constructs introduced as a result
Conclusion

• Semi-structured data best suited for data exchange

• “General” guidelines:
  – For quick, ad-hoc data analysis, use a “native” query language: SQL++, or AQL, or Xquery
  • Where “native” = how data is stored
  – Modern, advanced query processors like AsterixDB / SQL++ can process semi-structured data as efficiently as RDBMS
  – For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS