Lecture 15: SQL++ Wrapup
Find each country’s GDP

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

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

Error: Type mismatch!

Need to “unnest” the array
In General

Needs to be an array or dataset (i.e., iterable)

Object to be iterated on

These cannot evaluate to an array or dataset!

```
SELECT ...
FROM R AS x, S AS y
WHERE x.f1 = y.f2;
```

Need to “unnest” the array

CSE 414 - Spring 18
Unnesting collections

mydata

{"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"} ]}

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

Answer

{"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"}

Form cross product between each x and its x.B
Unnesting collections

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

```json
{"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"} ]}
```

**Answer**

Same as before

```
{"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"}
```
Find each country’s GDP

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

Answer

```
{ "name": "Albania", "gdp_total": "4100" }
{ "name": "Greece", "gdp_total": "101700" }
...
```
Return province and city names

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:

Error: Type mismatch!

"name": "Greece",
"province": [ ...
{"name": "Attiki",
 "city": [ {"name": "Athens"...}, {"name": "Pireus"...}, ...
 ...},
{"name": "Ipiros",
 "city": {"name": "Ioannia"...}
 ...}, ...
"

city is an array

city is an object
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) AS u
WHERE y.name="Greece";
Useful Functions

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

• Unnesting
• Nesting
• Grouping and aggregate
• Joins
• Multi-value join
We want:

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

Using LET syntax:

```sql
SELECT DISTINCT x.A, g AS Grp
FROM C AS x
LET g = (SELECT y.B FROM C AS y WHERE x.A = y.A)
```
Grouping and Aggregates

Count the number of elements in the F array for each A

\[
\text{SELECT } x.A, \text{ COLL\_COUNT}(x.F) \text{ AS cnt} \\
\text{FROM C AS } x
\]

These are NOT equivalent!

\[
\text{SELECT } x.A, \text{ COUNT(*) AS cnt} \\
\text{FROM C AS } x, x.F \text{ AS } y \\
\text{GROUP BY } x.A
\]

[[A:a1, F:[{B:b1}, {B:b2}],
{A:a2, F:[{B:b3}, {B:b4}, {B:null}],
{A:a3, F:[{B:b6}],
G:[{C:c1}]
G:[{C:c2},{C:c3}]]]}}
<table>
<thead>
<tr>
<th>Function</th>
<th>NULL</th>
<th>MISSING</th>
<th>Empty Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLL_COUNT</td>
<td>counted</td>
<td>counted</td>
<td>0</td>
</tr>
<tr>
<td>COLL_SUM</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MAX</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MIN</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_AVG</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_COUNT</td>
<td>not counted</td>
<td>not counted</td>
<td>0</td>
</tr>
<tr>
<td>ARRAY_SUM</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MAX</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MIN</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_AVG</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
</tbody>
</table>
Grouping and Aggregates

C

\[
\begin{align*}
{A: a1, F: [{B: b1}, {B: b2}], G: [{C: c1}]}, \\
{A: a2, F: [{B: b3}, {B: b4}, {B: null}], G: [ ]}, \\
{A: a3, F: [{B: b6}], G: [{C: c2}, {C: c3}]}\end{align*}
\]

Lesson:

Read the *#$@# manual!!

These are NOT equivalent!

SELECT x.A, COUNT(*) AS cnt
FROM C AS x, x.F AS y
GROUP BY x.A
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

Answer

[ {A:a1, B:b1, C:c1}, {A:a1, B:b1, C:c2}, {A:a2, B:b1, C:c1}, {A:a2, B:b1, C:c2} ]
Outer Joins

Two flat collection

\[
\text{coll1} \quad \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

\[
\text{coll2} \quad \{\{B:b1, C:c1\}, \{B:b1, C:c2\}, \{B:b3, C:c3\}\}
\]

\[
\text{SELECT} \ x.A, \ x.B, \ y.C \\
\text{FROM} \ \text{coll1} \ \text{AS} \ x \ \text{x \ RIGHT \ OUTER \ JOIN} \ \text{coll2} \ \text{AS} \ y \ \text{ON} \ x.B = y.B
\]

Answer

\[
\{\{A:a1, B:b1, C:c1\}, \{A:a1, B:b1, C:c2\}, \{A:a2, B:b1, C:c1\}, \{A:a2, B:b1, C:c2\}, \{B:b3, C:c3\}\}
\]
Ordering

```
coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

SELECT x.A, x.B
FROM coll AS x
ORDER BY x.A
```

Data type matters!

"90" > "8000" but
90 < 8000 !
Multi-Value Join

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

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

`split("MEX USA", " ") = ["MEX", "USA"]`
Behind the Scenes

Query Processing on NFNF 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:a1, F:\{B:b6\}, G:\{\{C:c2\},\{C:c3\}\}\}]

### 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:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}
]```

### Relational representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
</tr>
</tbody>
</table>
**Flattening SQL++ Queries**

**A nested collection**

coll =

\[
\begin{align*}
\{ & A:\text{a1}, F:\{ & B:b1, B:b2\}, G:\{ & C:c1\}\}, \\
\{ & A:\text{a2}, F:\{ & B:b3, B:b4, B:b5\}, G:\{ \}\}, \\
\{ & A:\text{a1}, F:\{ & B:b6\}, G:\{ & C:c2, C:c3\}\}\}
\end{align*}
\]

**Relational representation**

**Flattening SQL++ Queries**

```
SELECT x.A, y.B
FROM coll AS x, x.F AS y
WHERE x.A = "a1"
```
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:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Relational representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th></th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td></td>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td></td>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td></td>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

SQL++

SELECT x.A, y.B
FROM coll AS x, x.F AS y
WHERE x.A = “a1”

SQL

SELECT x.A, y.B
FROM coll AS x, F AS y
WHERE x.id = y.parent AND x.A = “a1”
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:a1, F:{{B:b6}}, G:{{C:c2},{C:c3}}}}

Relational representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
</tr>
</tbody>
</table>

SQL++

SELECT x.A, y.B
FROM coll AS x, x.F AS y
WHERE x.A = "a1"

SQL

SELECT x.A, y.B
FROM coll AS x, F AS y
WHERE x.id = y.parent AND x.A = "a1"

SELECT x.A, y.B
FROM coll AS x, x.F AS y, x.G AS z
WHERE y.B = z.C
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:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Relational representation

coll:

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
</tr>
</tbody>
</table>

F

<table>
<thead>
<tr>
<th>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

G

<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
</tr>
</tbody>
</table>

SQL++

SELECT x.A, y.B
FROM coll AS x, x.F AS y
WHERE x.A = “a1”

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

SQL

SELECT x.A, y.B
FROM coll AS x, F AS y
WHERE x.id = y.parent AND x.A = ‘a1’

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