Introduction to Database Systems  
CSE 414  
Lecture 15: SQL++ Wrapup  

Find each country’s GDP  

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

In General  

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

Unnesting collections  

mydata  

SELECT ...  
FROM mydata AS x, x.B AS y;  

Unnesting collections  

mydata  

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

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’;  

Error: Type mismatch!  

Need to "unnest" the array  

These cannot evaluate to an array or dataset!
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!

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";

Even better

SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
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

Grouping and Aggregates

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

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

These are NOT equivalent!
Grouping and Aggregates

<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>

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

Ordering

coll1 = [(A:a1, B:b1), (A:a1, B:b2), (A:a2, B:b1)]

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

Data type matters!

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

Multi-Value Join

coll1 =

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

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

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

Lesson:
Read the *$@# manual!!

Not equivalent!
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 = [(N1, F: [B1], G: [C1]),
        (N2, F: [B1, B2], G: [C1]),
        (N3, F: [B1], G: [C2, C3])]```

Relational representation

```
coll:
  id   A
  1    a1
  2    a2
  3    a1

  F  parent B
  1   1   B2
  2   2   B1
  2   2   B2
  2   2   B3

  G  parent C
  1   1   C1
  2   2   C2
  3   3   C3
```

SQL

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

Flattening SQL++ Queries

A nested collection

```
coll = [(N1, F: [B1], G: [C1]),
        (N2, F: [B1, B2], G: [C1]),
        (N3, F: [B1], G: [C2, C3])]```

Relational representation

```
coll:
  id   A
  1    a1
  2    a2
  3    a1

  F  parent B
  1   1   B2
  2   2   B1
  2   2   B2
  2   2   B3

  G  parent C
  1   1   C1
  2   2   C2
  3   3   C3
```

SQL

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

Flattening SQL++ Queries

A nested collection

```
coll = [(N1, F: [B1], G: [C1]),
        (N2, F: [B1, B2], G: [C1]),
        (N3, F: [B1], G: [C2, C3])]```

Relational representation

```
coll:
  id   A
  1    a1
  2    a2
  3    a1

  F  parent B
  1   1   B2
  2   2   B1
  2   2   B2
  2   2   B3

  G  parent C
  1   1   C1
  2   2   C2
  3   3   C3
```

SQL

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

Relational representation

coll

<table>
<thead>
<tr>
<th>ID</th>
<th>A</th>
<th>parent.B</th>
<th>parent.C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
<td>c2</td>
</tr>
<tr>
<td>2</td>
<td>a3</td>
<td>2</td>
<td>c3</td>
</tr>
<tr>
<td>3</td>
<td>a4</td>
<td>2</td>
<td>c4</td>
</tr>
<tr>
<td>2</td>
<td>a5</td>
<td>2</td>
<td>c5</td>
</tr>
<tr>
<td>3</td>
<td>a6</td>
<td>3</td>
<td>c6</td>
</tr>
</tbody>
</table>

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, 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

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 NNF 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