Introduction to Database Systems
CSE 414

Lecture 13: SQL++
Announcements
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

Can’t have repeated fields
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

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": "123456789"}
Open Types

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": "123456789"}
Types with Nested Collections

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
Dataset with Existing Key

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

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

{"name": "Alice"}
{"name": "Bob"}
...
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    myKey: uuid,
    Name : string,
    email: string?
}

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

{“name”: “Alice”}
{“name”: “Bob”}
…

Note: no myKey inserted as it is autogenerated
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
Example from HW5

mondial.adm is totally semi-structured:
{“mondial”: {“country”: [...], “continent”: [...], ..., “desert”: [...]} }

<table>
<thead>
<tr>
<th>country</th>
<th>continent</th>
<th>organization</th>
<th>sea</th>
<th>...</th>
<th>mountain</th>
<th>desert</th>
</tr>
</thead>
</table>
| [{“name”:”Albania”,...},
{“name”:”Greece”,...},
...] | ... | ... | ... | ... | ... | ... |

Nested objects!

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>...</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>...</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

USE myDB;
CREATE INDEX countryID
  ON country(`-car_code`) 
  TYPE BTREE;

Cannot index inside a nested collection

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

Country:

<table>
<thead>
<tr>
<th>-car_code</th>
<th>name</th>
<th>...</th>
<th>ethnicgroups</th>
<th>religions</th>
<th>...</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
<td>...</td>
<td>[ ... ]</td>
<td>[ ... ]</td>
<td>...</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>BG</td>
<td>Belgium</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•14</td>
</tr>
</tbody>
</table>
SQL++ Overview

SELECT ... 
FROM ... 
WHERE ... 
GROUP BY ... 
HAVING ... 
ORDER BY ...
Retrieve Everything

A collection of objects

1. Bind each object in world to x

2. Return mondial for each x

SELECT x.mondial FROM world AS x;

Answer

```json
{{
  "mondial":
  {
    "country": [
      Albania,
      Greece,
      ...
    ],
    "continent": [...],
    "organization": [...],
    ...
  }
}}
```
SELECT x.mondial AS ans FROM world AS x;

Answer

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

```
world
{
    "mondial":
    {
        "country": [{Albania}, {Greece} ...],
        "continent": [...],
        "organization": [...],
        ...
    }
}
```
Retrieve countries

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

Answer

```json
{"country": [{"Albania"}, {"Greece"}, ...], ...}
```
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;
SELECT x.mondial.country.name, c.gdp_total
FROM world AS x, country AS c
WHERE x.mondial.country.`-car_code` = c.`-car_code`;

"-car_code" is an illegal field name
Escape using ` ... `
Find each country’s GDP

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

Error: Type mismatch!

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

```
Need to “unnest” the array
```

world

```json
{{
  "mondial":
  {
    "country":
    [{
      "-car_code":"AL",
      "name":"Albania",
      ...,
    }, ...
    }, ...
  }
}}
```

country

```json
{{
  { 
    "-car_code":"AL",
    "gdp_total":4100,
    ...
  }, ...
}}
```
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;

Iterate over each x
and bind each object in x.B to y
Unnesting collections

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

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 UNNEST x.B AS y;

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

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

The problem:

```
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";
```

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

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

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" AND IS_ARRAY(z.city);
```
Return province and city names

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

The problem:

```json
{"mondial":
{"country": [{Albania}, {Greece}, ...],
"continent": [...],
"organization": [...],
...]
}
```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country AS y, y.province AS z,
(CASE WHEN IS_ARRAY(z.city) THEN z.city ELSE [z.city] END) AS u
WHERE y.name="Greece";
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";

Even better
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
We want:

```
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:

```
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

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}]}]

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

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

SELECT x.A, COUNT(*) AS cnt
FROM C AS x, x.F AS y
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>
Grouping and Aggregates

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

C

Lesson:
Read the *#$@# manual!!

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

These are NOT equivalent!
Joins

Two flat collection

\[
\text{coll1} = \begin{bmatrix}
\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}
\end{bmatrix}
\]

\[
\text{coll2} = \begin{bmatrix}
\{B:b1, C:c1\}, \{B:b1, C:c2\}, \{B:b3, C:c3\}
\end{bmatrix}
\]

Answer

\[
\begin{align*}
\text{SELECT} & \quad x.A, x.B, y.C \\
\text{FROM} & \quad \text{coll1 AS x, coll2 AS y} \\
\text{WHERE} & \quad x.B = y.B
\end{align*}
\]

\[
\begin{bmatrix}
\{A:a1, B:b1, C:c1\}, \\
\{A:a1, B:b1, C:c2\}, \\
\{A:a2, B:b1, C:c1\}, \\
\{A:a2, B:b1, C:c2\}
\end{bmatrix}
\]

\[
\begin{align*}
\text{SELECT} & \quad x.A, x.B, y.C \\
\text{FROM} & \quad \text{coll1 AS x JOIN coll2 AS y ON x.B = y.B}
\end{align*}
\]
Outer 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}]

SELECT x.A, x.B, y.C
FROM coll1 AS x RIGHT OUTER 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},
 {B:b3, C:c3}]}
Ordering

\[
\begin{array}{l}
\text{coll1} \quad \left[ \{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\} \right] \\
\end{array}
\]

\[
\text{SELECT} \ x.A, \ x.B \\
\text{FROM} \ \text{coll} \ \text{AS} \ x \\
\text{ORDER BY} \ x.A
\]

Data type matters!

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

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

```
["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}]]
```
**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>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td></td>
<td></td>
<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>

SELECT x.A, y.B
FROM coll AS x, x.F AS y
WHERE x.A = "a1"
Flattening SQL++ Queries

A nested collection

coll =

\[
\begin{array}{l}
\{A:a1, F: [{B:b1}, {B:b2}], G: [{C:c1}]\},
\{A:a2, F: [{B:b3}, {B:b4}, {B:b5}], G: []\},
\{A:a1, F: [{B:b6}], G: [{C:c2}, {C:c3}]\}\n\end{array}
\]

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”

SQL

SELECT x.A, y.B
FROM coll AS x, F AS y
WHERE x.id = y.parent AND x.A = “a1”
Flattening SQL++ Queries

A nested collection

Relational representation

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}]}]

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”

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"
```

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 y.B = z.C
```

```
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