Introduction to Data Management CSE 414

Lecture 14: SQL++

Announcements

- Midterm in class on Wednesday, 10/31
  - Covers everything (HW, WQ, lectures, sections, readings) up to beginning of NoSQL lectures.
  - One double-sided sheet of notes allowed
  - Remember to practice your SQL queries

- WQ5 and HW4 due Tuesday 10/30

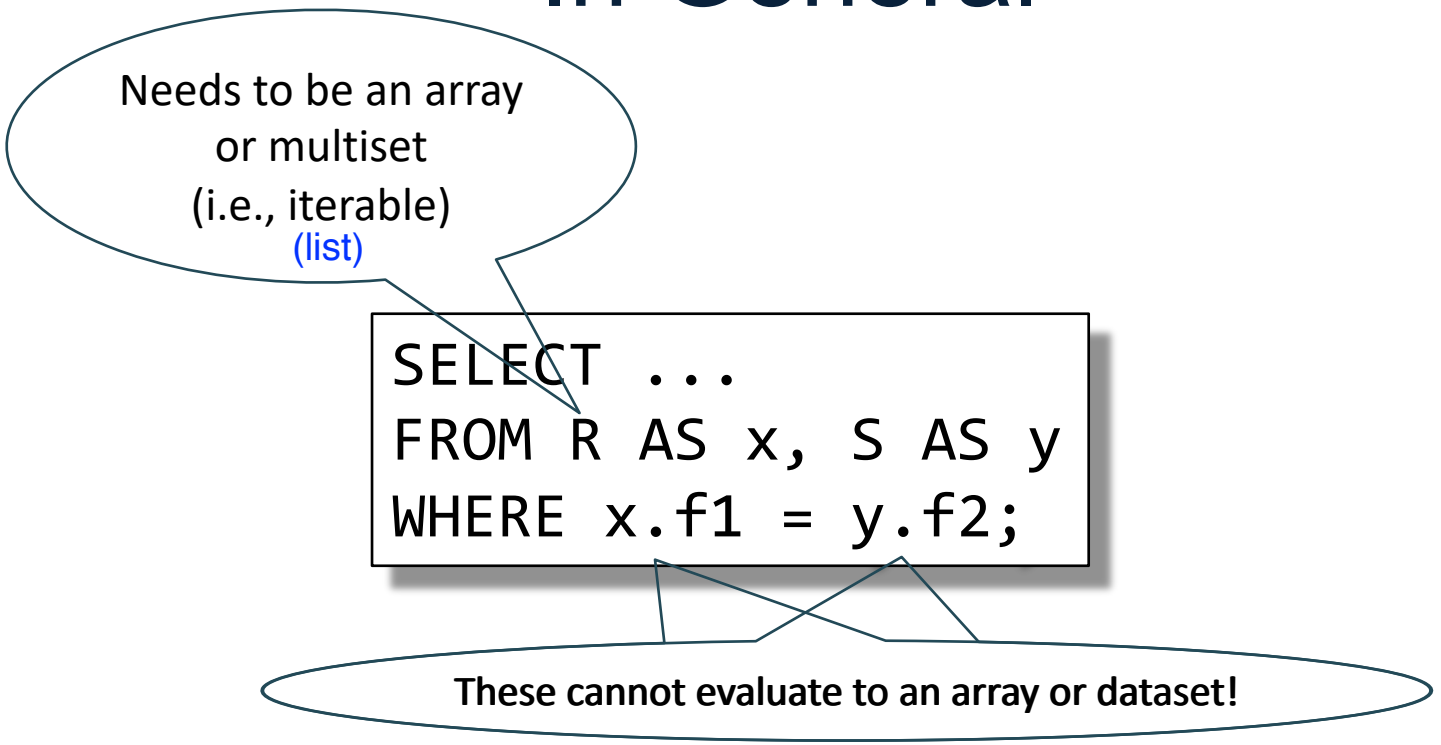
Types with Nested Collections

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name : string,
    phone: [string]
}
```

wont allow you to pass a parameter that is just a string; must be list

```
{"Name": "Carol", "phone": ["1234”]}
{"Name": "David", "phone": [“2345”, “6789”]}
{"Name": "Evan",  "phone": [ ]}
```

In General



Need to "unnest" the array

world

```
{{ {“mondial”:
    {“country”: [{Albania}, {Greece}, …],
     “continent”: […],
     “organization”: […],
      ...
      ...
     }
   }
}}
```

Return province and city names

world, x, y, and z MUST be lists of objects

```
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
OK

city is an object
ERROR

world

```
{{ {“mondial”:
    {“country”: [{Albania}, {Greece}, …],
     “continent”: […],
     “organization”: […],
      ...
      ...
     }
   }
}}
```

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);
```

The problem:

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

city is an array
OK

city is an object
ERROR

world

```
{{ {“mondial”:
    {“country”: [{Albania}, {Greece}, …],
     “continent”: […],
     “organization”: […],
      ...
      ...
     }
   }
}}
```

Return province and city names

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
WHERE y.name="Greece" AND NOT IS_ARRAY(z.city);
```

The problem:

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

city is an array ERROR

city is an object OK

world

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

Return province and city names

```
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";
```

Get both!

world

```
{{ {“mondial”:
    {“country”: [{Albania}, {Greece}, …],
     “continent”: […],
     “organization”: […],
      ...
      ...
     }
   }
}}
```

Return province and city names

```
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 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

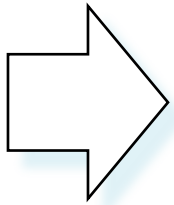
Useful Paradigms

- Unnesting
- Nesting
- Grouping and aggregate
- Joins
- Splitting

Nesting

C

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



We want:

```
[{A:a1, Grp:[{b1, b2}]},
 {A:a2, Grp:[{b1}]}]
```

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

both are equivalent, but let is a bit cleaner

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)
```

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}]}]
```

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}]}]

Nested Relational Algebra

$Unnest_F$(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}]}]

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}]}]

Nested Relational Algebra

$Unnest_F$(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}]}]

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}]}]
```

Nested Relational Algebra

$Unnest_F$(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}]}]
```

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

Refers to relations defined on the left

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}]}]
```

Nested Relational Algebra

$Unnest_F(coll)$ =

```
[{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}]}]
```

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

=

SQL++

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

NOT a join; just have one table; Unnesting

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}]}]
```

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

unnesting with respect to G. Note that for the 2nd collecti
where G = [], NO output is produced.

Nested Relational Algebra

$Unnest_G(coll)$ =

```
[{A:a1, F:[{B:b1},{B:b2}], C:c1},
 {A:a3, F:[{B:b6}], C:c2},
 {A:a3, F:[{B:b6}], C:c3]}
```

SQL++

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

for each x in coll, for each z in x.G …

Nesting (like group-by)

A flat collection

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

Nested Relational Algebra

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

$Nest_B$(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, g as GRP
FROM coll x
LET g =  (SELECT y.B FROM coll 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 G array for each A

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

COLL_COUNT counts the number of elements in each collection

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

forces us to list thru each A-G pair

These are NOT equivalent!

Initial query is already nested… so we have to unnest it first then collect it under x.A

Grouping and Aggregates

Function	NULL	MISSING	Empty Collection
COLL_COUNT	counted	counted	0
COLL_SUM	returns NULL	returns NULL	returns NULL
COLL_MAX	returns NULL	returns NULL	returns NULL
COLL_MIN	returns NULL	returns NULL	returns NULL
COLL_AVG	returns NULL	returns NULL	returns NULL
ARRAY_COUNT	not counted	not counted	0
ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

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 G array for each A

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

query 1 output:

{"cnt": 1, "A":a1 }
{"cnt": 0, "A":a2 }
{"cnt": 2, "A":a3 }

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

query 2 output

{"cnt": 1, "A":a1 }
{"cnt": 2, "A":a3 }

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}]
```

on flat tables, joins are exactly like they are in SQL

```
SELECT x.A, x.B, y.C
FROM coll1 AS x, coll2 AS y
WHERE 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}]
```

```
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: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}]
```

uses missing rather than NULL

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
```

Be careful of data type

Splitting

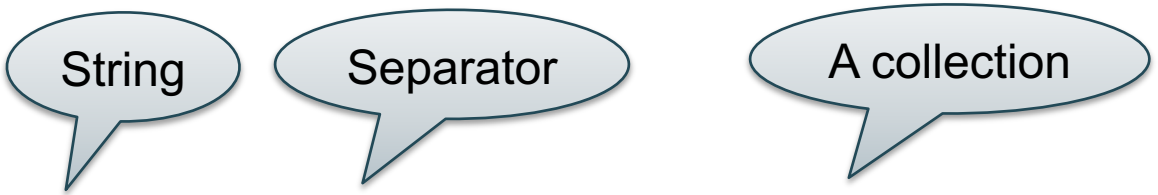
- Nesting allows us to keep collections of references in the place of a single value.
- Sometimes these references are kept in a single string:
  - The reference is a string of keys separated by space
  - Need to use split(string, separator) to split it into a collection of foreign keys

Splitting

river

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

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




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


creates a list out of the string!

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}]}]
```

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:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	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}]}]
```

SQL++

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

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	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}]}]
```

SQL++

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

nested, have to unnest

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

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

have to join

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}]}]
```

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
```

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

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

```
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}]}]
```

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
WHERE y.B = z.C
```

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

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