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`~~`}`

Can’t have repeated fields

- 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

```
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?
}
```

```
{"name": "Alice"}
{"name": "Bob"}
…
```

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

Dataset with Auto Generated Key

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   myKey: uuid,
   Name : string,
   email: string?
}
```

```
{"name": "Alice"}
{"name": "Bob"}
…
```

Note: no myKey inserted as it is autogenerated

```
USE myDB;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey 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":[...]}}
```

country	continent	organization	sea	...	mountain	desert
[{"name":"Albania",...}, {"name":"Greece",...}, ...]	...	...	...		...	...

Nested objects!

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

Country:

-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...	...	...	...			

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

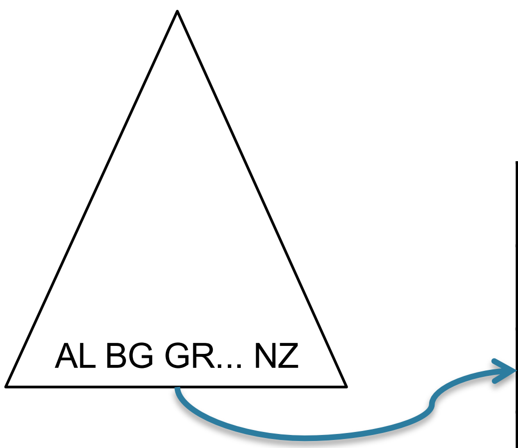
- Will discuss how they help later in the quarter

Indexes

Cannot index inside a nested collection

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

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



Country:

-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...	...	...	...			
BG	Belgium	...				
...						

SQL++ Overview

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
```

Retrieve Everything

world

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

A collection of objects

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

2. Return mondial for each x

1. Bind each object in world to x

Answer

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

Retrieve Everything

world

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

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

Answer

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

world

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

Retrieve countries

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

Answer

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

world

```
{{ {“mondial”:
    {“country”:
     [{“-car_code”:"AL”, …}
       {“name”:”Albania”}, …
       ], ...
    }, ...
}}
```

country

```
{{ { “-car_code”:“AL”,
      “gdp_total”:4100,
       ...
    }, ...
}}
```

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`;
```

Find each country's GDP

world

```
{{ {“mondial”:
    {“country”:
     [{“-car_code”:"AL”, …}
      {“name”:”Albania”}, …
      ], ...
    }, ...
}}
```

country

```
{{ { “-car_code”:“AL”,
     “gdp_total”:4100,
     ...
   }, ...
}}
```

“-car_code” is an illegal field name
Escape using ` ... `

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

Find each country's GDP

world

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

country

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

```
SELECT x.mondiao.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

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

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;
```

Form cross product between each x and its x.B

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"}
```

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;
```

Same as before

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"}
```

world

```
{{ {“mondial”:
    {“country”:
     [{“-car_code”:"AL”, …}
      {“name”:”Albania”}, …
      ], ...
    }, ...
}}
```

country

```
{{ { “-car_code”:“AL”,
     “gdp_total”:4100,
     ...
   }, ...
}}
```

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" }
...
```

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";
```

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

Return province and city names

world

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

```
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

city is an object

Return province and city names

world

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

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

city is an object

Return province and city names

world

```
{{ {"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";
```

array

Get both!

Return province and city names

world

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

```
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

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
```

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

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}]}]
```

(

```
SE
FR
```

A

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}]
```

```
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}]
```

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

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
```

String

Separator

A collection

```
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}]}]
```

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”
```

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 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}]}]
```

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, 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
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 *<u>data exchange</u>*
- “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