Database Systems
CSE 414

Lecture 16: JSON and SQL++
(mostly not in textbook)

Announcement

- Midterm graded
  - to be handed back in section
  - answers were okay on problems 1, 3, 4
  - problem 2 caused a lot of difficulty
    - expect a final question on cost based optimization
- HW5 will be posted tonight: due May 9
  - uses AsterixDB
- Section on installing & using AsterixDB
  - get through problem 1 tomorrow

NoSQL (cont)

Google Cloud Datastore demo

Kind

Ticker

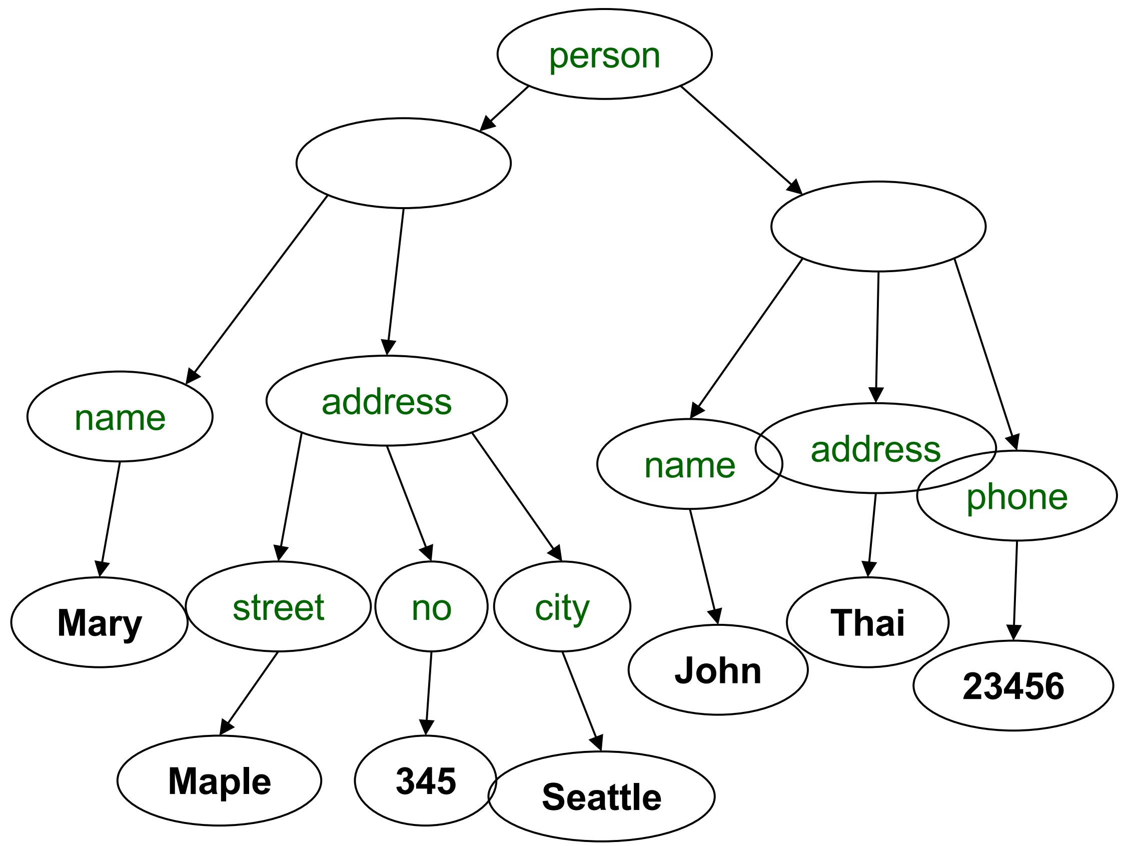
Filter entities

Name/ID	CIKNo	CUSIP	CompanyName
name=A	MDAwMTA5MDg3Mg==	MDA4NDZVMTAx	QWdpbGVudCBUZWNobm9sb2dpZXMgSW5j
name=AA	MDAwMTY3NTE0OQ==	MDEzODcyMTA2	QWxjb2EgQ29ycA==
name=AAL	MDAwMDAwNjIwMQ==	MDIzNzZSMTAy	QW1lcmljYW4gQWlybGluZXMgR3JvdXAgSW5j
name=AAP	MDAwMTE1ODQ0OQ==	MDA3NTFZMTA2	QWR2YW5jZSBBdXRvIFBhcnRzLCBJbmMu
name=AAPL	MDAwMDMyMDE5Mw==	MDM3ODMzMTAw	QXBwbGUgSW5jLg==
name=ABBV	MDAwMTU1MTE1Mg==	MDAyODdZMTA5	QWJiVmllIEluYw==
name=ABC	MDAwMTE0MDg1OQ==	MDMwNzNFMTA1	QW1lcmlzb3VyY2VCZXJnZW4gQ29ycC4=
name=ABT	MDAwMDAwMTgwMA==	MDAyODI0MTAw	QWJib3R0IExhYm9yYXRvcmllcw==
name=ACN	MDAwMTQ2NzM3Mw==	RzExNTFDMTAx	QWNjZW50dXJlIFBsYw==
name=ADBE	MDAwMDc5NjM0Mw==	MDA3MjRGMTAx	QWRvYmUgU3lzdGVtcyBJbmNvcnBvcmF0ZWQ=
name=ADI	MDAwMDAwNjI4MQ==	MDMyNjU0MTA1	QW5hbG9nIERldmljZXMsIEluYy4=

JSON (cont.)

JSON Semantics: a Tree !

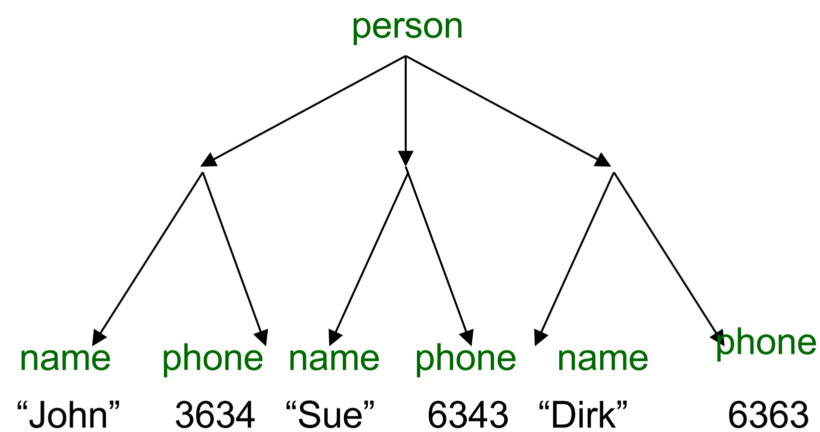
```
{“person”:
  [ {“name”: “Mary”,
     “address”:
        {“street”:“Maple”,
          “no”:345,
          “city”: “Seattle”}},
    {“name”: “John”,
     “address”: “Thailand”,
     “phone”:2345678}}
  ]
}
```



JSON Data

- JSON is self-describing
- Schema elements become part of the data
  - Relational schema: person(name,phone)
  - In JSON "person", "name", "phone" are part of the data, and are repeated many times
- Consequence: JSON is much more flexible
- JSON = semistructured data

Mapping Relational Data to JSON



Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{“person”:
     [{“name”: “John”, “phone”:3634},
      {“name”: “Sue”,  ”phone”:6343},
      {“name”: “Dirk”,  ”phone”:6383}
     ]
}
```

Mapping Relational Data to JSON

May inline foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

personName	date	product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

```
{"Person":
     [{"name": "John",
       "phone":3646,
       "Orders":[{"date":2002,
                  "product":"Gizmo"},
                 {"date":2004,
                  "product":"Gadget"}
                ]
      },
      {"name": "Sue",
       "phone":6343,
       "Orders":[{"date":2002,
                  "product":"Gadget"}
                ]
      }
     ]
}
```

JSON =Semi-structured Data (1/3)

- Missing attributes:

```
{"person":
    [{"name":"John", "phone":1234},
     {"name":"Joe"}]
}
```

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	-

JSON =Semi-structured Data (2/3)

- Repeated attributes

```
{"person":
   [{"name":"John", "phone":1234},
    {"name":"Mary", "phone":[1234,5678]}]
}
```

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???

JSON =Semi-structured Data (3/3)

- Attributes with different types in different objects

```
{“person”:
    [{“name”:”Sue”, “phone”:3456},
     {“name”:{“first”:”John”,”last”:”Smith”},”phone”:2345}
    ]
}
```

Structured name !

- Nested collections
- Heterogeneous collections

Discussion

- *Data exchange formats*
  - well suited for exchanging data between apps
  - XML, JSON, Protobuf
- Increasingly, some systems use them as a data model:
  - SQL Server supports for XML-valued relations
  - CouchBase, Mongodb: JSON as data model
  - Dremel (BigQuery): Protobuf as data model

Query Languages for SS Data

- XML: XPath, XQuery (see end of lecture, textbook)
  - Supported inside many RDBMS (SQL Server, DB2, Oracle)
  - Several standalone XPath/XQuery engines
- Protobuf: used internally by google, and externally in BigQuery. similar to compiled JSON
- JSON:
  - CouchBase: N1QL
  - MongoDB: has a pattern-based language
  - JSONiq http://www.jsoniq.org/
  - AsterixDB: AQL and SQL++

AsterixDB

AsterixDB

- NoSQL database system (document store)
- Developed at UC Irvine
  - Now an Apache project
- Designed to be installed on a cluster
  - multiple machines (nodes) together implement the DBMS
  - allows scale to much larger amounts of data
- Weak support for multi-node transactions
- Good support for multi-node **queries**

AsterixDB (cont.)

- Data is **partitioned** over nodes by primary key
  - queries involve not only disk but also network I/O
- Supports advanced queries
  - joins
  - nested queries
  - grouping and aggregation
- No statistics maintained yet (per docs)
  - may need more hints to get good performance
  - expect this to improve

AQL and SQL++

- Asterix own query language is AQL
  – based on XQuery (for XML)

- SQL++
  – SQL-like syntax for AQL
  – more familiar to database users

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {“Name”: “Alice”, “age”: 40}
  - Fields must be distinct:
    {“Name”: “Alice”, “age”: 40, ~~“age”:50~~}

Can’t have repeated fields

- Arrays:
  - [1, 3, “Fred”, 2, 9]
  - Note: can be heterogeneous
- Bags:
  - {{1, 3, “Fred”, “Fred”, 2, 9}}

Examples

Try these queries yourself:

```
SELECT age FROM [ {'name': 'Alice', 'age': ['30', '50']} ] x;
```

```
SELECT age FROM {{ {'name': 'Alice', 'age': ['30', '50']} }} x;
```

```
-- error
SELECT age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

Datatypes

- Supports SQL / JSON data type:
  - boolean, integer, float (various precisions), null
- Some SQL types not in JSON:
  - date, time, interval
- Some new types:
  - geometry (point, line, …)
  - UUID = universally unique identifier
    (systems generated, globally unique key)

Null vs Missing

- {“age”: null} = the value NULL (like in SQL)
- {“age”: missing} = { } = really missing

```
SELECT x.b FROM [{'a':1, 'b':2}, {'a':3}] x;
```

```
{ "b": 2 }
{  }
```

```
SELECT x.b FROM [{'a':1, 'b':2}, {'a':3, 'b':missing }] x;
```

```
{ "b": 2 }
{  }
```

SQL++ Overview

- Data definition language:
  - Dataverse (= database)
  - Dataset (= table)
    - each row uses a declared Type
  - Types
    - declares the required parts
    - can allow for extra data (open vs closed types)
  - Indexes
- Query language: select-from-where

Dataverse

A Dataverse is a Database

- CREATE DATAVERSE lec16
- CREATE DATAVERSE lec16 IF NOT EXISTS

- DROP DATAVERSE lec16
- DROP DATAVERSE lec16 IF EXISTS

- USE lec16

Type

- Defines the schema of a collection
- It lists all *required* fields
- Fields followed by ? are *optional*
- CLOSED type = no other fields allowed
- OPEN type = other fields allowed

Closed Types

```
USE lec16;
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", “age”:35, ~~"phone": "123456789"~~}

Open Types

```
USE lec16;
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”:35, "phone": "123456789"}

Types with Nested Collections

```
USE lec16;
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 be declared “autogenerated” if UUID
  - (SQL systems usually support autoincremented unique integer IDs)

Dataset with Existing Key

```
USE lec16;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   email: string?
}
```

{“Name”: “Alice”}
{“Name”: “Bob”}
…

```
USE lec16;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

Dataset with Auto Generated Key

```
USE lec16;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   myKey: uuid,
   Name : string,
   email: string?
}
```

```
USE lec16;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;
```

{“Name”: “Alice”}
{“Name”: “Bob”}
…

Note: no myKey since it will be autogenerated

Discussion of NFNF

- 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 semistructured:
{“mondial”: {“country”: [...], “continent”:[...], ..., “desert”:[...]}}

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

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:
  - 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

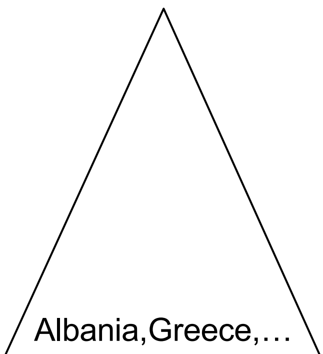
as you already know…

Indexes

Cannot index inside a nested collection

```
USE lec16;
CREATE INDEX countryID
    ON country(name)
    TYPE BTREE;
```

```
USE lec16;
CREATE INDEX cityname
    ON country(city.name)
    TYPE BTREE;
```



Country:

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

Asterix Data Model (ADM)

- ADM is an extension of JSON
- Objects:
  - {"Name": "Alice", "age": 40}
  - Fields must be distinct:
    {"Name": "Alice", "age": 40, ~~"age":50~~}

Can't have repeated fields

- Arrays:
  - [1, 3, "Fred", 2, 9]
  - Note: can be heterogeneous
- Bags:
  - {{1, 3, "Fred", "Fred", 2, 9}}

Examples

Try these queries yourself:

```
SELECT age FROM [ {'name': 'Alice', 'age': ['30', '50']} ] x;
```

~> {"age": ["30", "50"]}

```
SELECT age FROM {{ {'name': 'Alice', 'age': ['30', '50']} }} x;
```

~> {"age": ["30", "50"]}

```
-- error
SELECT age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

SQL++ Overview

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```

```
[{"mondial":
    {"country": [ country1, country2, …],
     "continent": […],
      "organization": […],
      ...
      ...
}]
```

Retrieve Everything

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

Answer

```
{"mondial":
      {"country": [ country1, country2, …],
      "continent": […],
      "organization": […],
      ...
      ...
}
```

Retrieve countries

```
[{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}]
```

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

Answer

```
{"country": [ country1, country2, …]}
```

Retrieve countries, one by one

```
[{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}]
```

```
SELECT y as country FROM world x, x.mondial.country y;
```

Answer

```
{"country": country1}
{"country": country2}
...
```

Escape characters

```
[{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}]
```

"-car_code" illegal field
Use ` ... `

```
SELECT y.`-car_code` as code , y.name as name
FROM world x, x.mondial.country y order by y.name;
```

Answer

```
{"code": "AFG", "name": "Afganistan"}
{"code": "AL", "name": "Albania"}
...
```

Nested Collections

- If the value of attribute B is some other collection, then we can simply iterate over it

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

x.B is a collection

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

```
{"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"}
```

Heterogeneous Collections

```
[{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}]
```

Runtime error

```
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:

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

city is an array

city is an object

Heterogeneous Collections

```
[{"mondial":
   {"country": [ country1, country2, …],
    "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);
```

Just the arrays

The problem:

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

Heterogeneous Collections

```
[{"mondial":
   {"country": [ country1, country2, …],
    "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);
```

Just the objects

The problem:

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

Heterogeneous Collections

```
[{"mondial":
   {"country": [ country1, country2, …],
    "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 is_array(z.city)
            THEN z.city ELSE [z.city] END) u
WHERE  y.name='Greece';
```

Get both!

The problem:

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

Heterogeneous Collections

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}
```

T

```
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)  u
WHERE  y.name='Greece';
```

Even better

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

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
- Group-by / aggregate
- Join
- Multi-value join

Basic Unnesting

- An array: [a, b, c]
- A nested array: arr = [[a, b], [], [b, c, d]]
- Unnest(arr) = [a, b, b, c, d]

```
SELECT y
FROM arr x, x y
```

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}]}]
```

New RA expression

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}]
```

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

SQL++

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

Nesting (like group-by)

A flat collection

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

new RA expression

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)
```

Group-by / Aggregate

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}]}]
```

Count the number of elements in the F collection

```
SELECT x.A, coll_count(x.F) as cnt
FROM coll x
```

```
SELECT x.A, count(*) as cnt
FROM coll x, x.F y
GROUP BY x.A
```

These are NOT equivalent!
(Why?)

Group-by / Aggregate

A flat collection

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

```
SELECT DISTINCT x.A, coll_count(g) as cnt
FROM coll x
LET g =  (SELECT y.B FROM coll y WHERE x.A = y.A)
```

```
SELECT x.A, count(*) as cnt
FROM coll x
GROUP BY x.A
```

Are these equivalent?

Join

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 x, coll2 y
WHERE x.B = y.B
```

Multi-Value Join

- A many-to-one relationship should have one foreign key, from “many” to “one”
  – each of the “many” points to the same “one”
- Sometimes people represent it in the opposite direction, from “one” to “many”:
  – Ex: list of employees of a manager
  – reference could be space-separated string of keys
  – need to use split(string, separator) to split it into a collection of foreign keys

Multi-Value Join

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

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

String Separator

split(“MEX USA”, “ “) =
[“MEX”, “USA”]

Behind the Scenes

Query Processing on NFNF data:

- Option 1: give up on query plans
- Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan

You can apply the second approach yourself, to work with SS data using a familiar RDBMS

- for data analysis, this may be more efficient until SS DBMSs catch up to RDBMSs

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 x, x.F y
WHERE x.A = ‘a1’
```

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

Flat 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 x, F y
WHERE x.id = y.parent and x.A = ‘a1’
```

```
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent and x.id = z.parent
          and y.B = z.C
```