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

<table>
<thead>
<tr>
<th>Kind</th>
<th>Ticker</th>
<th>CIKNo</th>
<th>CUSIP</th>
<th>CompanyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>name=A</td>
<td></td>
<td>MDAwMTA5MDg3Mg==</td>
<td>MDA4NDZVMTAx</td>
<td>QWdpbGVudCBUZNobm9sb2dpZXMgSW5j</td>
</tr>
<tr>
<td>name=AA</td>
<td></td>
<td>MDAwMTY3NTE0Q0==</td>
<td>MDEzODcyMTA2</td>
<td>QWxjb2EqQ29ycA==</td>
</tr>
<tr>
<td>name=AAL</td>
<td></td>
<td>MDAwMDAwNljwMQ==</td>
<td>MDizNzZSMTAy</td>
<td>QW1lcmlijYW4gQWlybGluZXMgR3JvdXAgSW5j</td>
</tr>
<tr>
<td>name=AAP</td>
<td></td>
<td>MDAwMTE1ODQ0Q0==</td>
<td>MDA3NTFZMTA2</td>
<td>QWR2YW5jZSB8dXRv1FBhcnRzLCBjbmMu</td>
</tr>
<tr>
<td>name=AAPL</td>
<td></td>
<td>MDAwMDMyMDE5Mw==</td>
<td>MDM3ODMzMTAw</td>
<td>QXBwbGUgSW5jLg==</td>
</tr>
<tr>
<td>name=ABBV</td>
<td></td>
<td>MDAwMTU1MTE1Mg==</td>
<td>MDAyODdZMTA5</td>
<td>QWJiVmllIElUYw==</td>
</tr>
<tr>
<td>name=ABC</td>
<td></td>
<td>MDAwMTE0MDg1Q0==</td>
<td>MDMwNzNFMTA1</td>
<td>QW1lcmIzb3VyY2VCZXJnZW4gQ29ycC4=</td>
</tr>
<tr>
<td>name=ABT</td>
<td></td>
<td>MDAwMDAwMTgwMA==</td>
<td>MDAyODI0MTAw</td>
<td>QWJib3R0iExhYm9yYXRvcmIlcw==</td>
</tr>
<tr>
<td>name=ACN</td>
<td></td>
<td>MDAwMTQ2NzM3Mw==</td>
<td>RzExNTFDMTAx</td>
<td>QWNjZW50dXJJIFBSyW==</td>
</tr>
<tr>
<td>name=ADBE</td>
<td></td>
<td>MDAwMDc5NjM0Mw==</td>
<td>MDA3MjRGMTAx</td>
<td>QWRVymUgU3IzclGytcyBJbmNvcnBvcmF0ZWQ=</td>
</tr>
<tr>
<td>name=ADI</td>
<td></td>
<td>MDAwMDAwNlj4MQ==</td>
<td>MDMyNlj0MTA1</td>
<td>QW5hbg9niERidmlljZXMsiElUYy4=</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
</tr>
</tbody>
</table>

```json
{
  "person": [
    {
      "name": "John",
      "phone": 3634
    },
    {
      "name": "Sue",
      "phone": 6343
    },
    {
      "name": "Dirk",
      "phone": 6363
    }
  ]
}
```
Mapping Relational Data to JSON

May inline foreign keys

Person

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
</tbody>
</table>

Orders

<table>
<thead>
<tr>
<th>personName</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2002</td>
<td>Gizmo</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
<td>Gadget</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
<td>Gadget</td>
</tr>
</tbody>
</table>

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

• Could represent in a table with nulls

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1234</td>
</tr>
<tr>
<td>Joe</td>
<td>-</td>
</tr>
</tbody>
</table>
JSON = Semi-structured Data (2/3)

• Repeated attributes

{“person”: 
{“name”:”John”, “phone”:1234},
{“name”:”Mary”, “phone”:[1234,5678]}
}

• Impossible in one table:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>2345</td>
</tr>
<tr>
<td></td>
<td>3456</td>
</tr>
</tbody>
</table>
JSON = Semi-structured Data (3/3)

- Attributes with different types in different objects
  
  ```json
  {"person":
   [{"name":"Sue", "phone":3456},
    {"name":{"first":"John","last":"Smith"},"phone":2345}]
  }
  ```

- Nested collections
- Heterogeneous collections

Structured name!
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
  – 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}

• 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
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?
}

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

{“Name”: “Alice”}
{“Name”: “Bob”}
…
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
**Indexes**

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

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

Cannot index inside a nested collection

---

**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></td>
</tr>
</tbody>
</table>
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\}
• Arrays:
  – [1, 3, “Fred”, 2, 9]
  – Note: can be heterogeneous
• Bags:
  – \{\{1, 3, “Fred”, “Fred”, 2, 9\}\}

Can’t have repeated fields
Examples

Try these queries yourself:

```sql
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

```sql
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```
Retrieve Everything

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

Answer

```json
[
  {
    "mondial":
    {
      "country": [ country1, country2, ...],
      "continent": [...],
      "organization": [...],
      ...
    }
  }
]
```
Retrieve countries

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

Answer

```json
{{"country": [ country1, country2, ...]}
```
Retrieve countries, one by one

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

Answer

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

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

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"
}
...

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

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

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

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

{x.B is a collection}
Heterogeneous Collections

The problem:

```sql
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';
```

city is an array

city is an object
Heterogeneous Collections

```sql
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:

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

Just the arrays

```json
{
  "province": [ ...
    {
      "name": "Attiki",
      "city": [ {
        "name": "Athens"}, {
        "name": "Pireus"}, ...
      }, ...
    },
    {
      "name": "Ipiros",
      "city": {
        "name": "Ioanna"...
      }
    }
  ...
}
```

45
Heterogeneous Collections

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:

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

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';

The problem:

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

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'
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: \( \text{arr} = [[a, b], [], [b, c, d]] \)
• \( \text{Unnest}(\text{arr}) = [a, b, b, c, d] \)

\[
\text{SELECT y} \\
\text{FROM arr x, x y}
\]
Unnesting Specific Field

A nested collection

\[
\text{coll} = \\
\begin{cases}
\{A:a1, F:[\{B:b1\},\{B:b2\}], G:[\{C:c1\}]\}, \\
\{A:a2, F:[\{B:b3\},\{B:b4\},\{B:b5\}], G:[]\}, \\
\{A:a3, F:[\{B:b6\}], G:[\{C:c2\},\{C:c3\}]\}\end{cases}
\]

\[
\text{Unnest}_F(\text{coll}) = \\
\begin{cases}
\{A:a1, B:b1, G:[\{C:c1\}]\}, \\
\{A:a1, B:b2, G:[\{C:c1\}]\}, \\
\{A:a2, B:b3, G:[]\}, \\
\{A:a2, B:b4, G:[]\}, \\
\{A:a2, B:b5, G:[]\}, \\
\{A:a3, B:b6, G:[\{C:c2\},\{C:c3\}]\}\end{cases}
\]

\[
\text{Unnest}_G(\text{coll}) = \\
\begin{cases}
\{A:a1, F:[\{B:b1\},\{B:b2\}], C:c1\}, \\
\{A:a3, F:[\{B:b6\}], C:c2\}, \\
\{A:a3, F:[\{B:b6\}], C:c3\}\end{cases}
\]

SELECT x.A, y.B, x.D FROM coll x, x.F y

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

New RA expression
Nesting (like group-by)

A flat collection

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

\[
\text{Nest}_A(\text{coll}) = \\
\{\{A:a1, \text{GRP}:\{B:b1, B:b2\}\}, \{A:a2, \text{GRP}:\{B:b2\}\}\}
\]

\[
\text{Nest}_B(\text{coll}) = \\
\{\{B:b1, \text{GRP}:\{A:a1, A:a2\}\}, \{B:b2, \text{GRP}:\{A:a1\}\}\}
\]

\[
\text{SELECT DISTINCT x.A,} \\
(\text{SELECT y.B FROM coll y WHERE x.A = y.A}) \text{ as GRP} \\
\text{FROM coll x}
\]

\[
\text{SELECT DISTINCT x.A, g as GRP} \\
\text{FROM coll x} \\
\text{LET g = (SELECT y.B FROM coll y WHERE x.A = y.A)}
\]

new RA expression
Group-by / Aggregate

A nested collection

\[
\text{coll} = \\
\left\{ \\
\{A:a1, F:\{B:b1\}, G:\{C:c1\}\}, \\
\{A:a2, F:\{B:b3\}, G:\{C:c2\}, G:\{C:c3\}\}, \\
\{A:a3, F:\{B:b6\}, G:\{C:c2\}, G:\{C:c3\}\}\right\}
\]

Count the number of elements in the F collection

\[
\text{SELECT} \ x.A, \ \text{coll\_count}(x.F) \ \text{as} \ cnt \\
\text{FROM} \ coll \ x
\]

These are NOT equivalent! (Why?)

\[
\text{SELECT} \ x.A, \ \text{count}(*) \ \text{as} \ cnt \\
\text{FROM} \ coll \ x, \ x.F \ y \\
\text{GROUP BY} \ x.A
\]
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

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

\[
\text{coll} = \\
[\{A:a1, F: [\{B:b1\}, \{B:b2\}], G: [\{C:c1\}]\}, \\
\{A:a2, F: [\{B:b3\}, \{B:b4\}, \{B:b5\}], G: [\} \}, \\
\{A:a1, F: [\{B:b6\}], G: [\{C:c2\}, \{C:c3\}]\}]
\]

Flat Representation

\[
\begin{array}{c|c}
\text{id} & \text{A} \\
\hline
1 & a1 \\
2 & a2 \\
3 & a1 \\
\end{array}
\quad
\begin{array}{c|c}
\text{parent} & \text{B} \\
\hline
1 & b1 \\
1 & b2 \\
2 & b3 \\
2 & b4 \\
2 & b5 \\
3 & b6 \\
\end{array}
\quad
\begin{array}{c|c}
\text{parent} & \text{C} \\
\hline
1 & c1 \\
3 & c2 \\
3 & c3 \\
\end{array}
\]

SQL++

\[
\text{SELECT x.A, y.B} \\
\text{FROM coll x, x.F y} \\
\text{WHERE x.A = 'a1'}
\]

\[
\text{SELECT x.A, y.B} \\
\text{FROM coll x, x.F y} \\
\text{WHERE y.B = z.C}
\]

SQL

\[
\text{SELECT x.A, y.B} \\
\text{FROM coll x, F y} \\
\text{WHERE x.id = y.parent and x.A = 'a1'}
\]

\[
\text{SELECT x.A, y.B} \\
\text{FROM coll x, F y, G z} \\
\text{WHERE x.id = y.parent and x.id = z.parent and y.B = z.C}
\]