Database Systems
CSE 514

Lecture 7: JSon and SQL++
NoSQL Motivation

• Originally motivated by Web 2.0 applications

• Goal is to scale simple OLTP-style workloads to thousands or millions of users (in class: OLTP v.s. OLAP)

• Users are doing both updates and reads
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

<table>
<thead>
<tr>
<th>Person</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>Sue</td>
<td>6343</td>
</tr>
<tr>
<td>Dirk</td>
<td>Dirk</td>
<td>6363</td>
</tr>
</tbody>
</table>

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

```json
{
  "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
Discussion

• *Data exchange formats*
  – Ideally 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 textbook)
  – Supported inside many RDBMS (SQL Server, DB2, Oracle)
  – Several standalone XPath/XQuery engines
• Protobuf: SQL-ish language (Dremel) used internally by google, and externally in BigQuery
• JSON:
  – CouchBase: N1QL
  – MongoDB: has a pattern-based language
  – JSONiq http://www.jsoniq.org/
AsterixDB and SQL++

• AsterixDB
  – No-SQL database system
  – Developed at UC Irvine
  – Now an Apache project
  – Own query language: AsterixQL or AQL, based on XQuery

• SQL++
  – SQL-like syntax for AsterixQL
Asterix Data Model (ADM)

• 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

• Multisets:
  – {{1, 3, “Fred”, 2, 9}}
Examples

Try these queries:

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;
Examples

Try this query:

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

Answer:

```
{ "x": { "name": "Alice", "age": [ "30", "50" ] } }
{ "x": { "name": "Dave", "age": [ "30" ] } }
```
Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, ...), date, time, etc

- UUID = universally unique identifier
  Use it as a systems generated unique key
Null v.s. Missing

• {"age": null} = the value NULL (like in SQL represent an unknown value)
• {"age": missing} = {} = really missing (indicates that a name-value pair is missing from an object)

SELECT Age FROM [{ 'Age': null }] x;
Result: { "Age": null }

SELECT Age FROM [{ 'Age': missing }] x;
Result: {}
SQL++ Overview

• DDL: create a
  – Dataverse
  – Type
  – Dataset
  – Index

• DML: select-from-where
A Dataverse is a Database

- CREATE DATaverse lec7
- CREATE DATaverse lec7 IF NOT EXISTS

- DROP DATaverse lec7
- DROP DATaverse lec7 IF EXISTS

- USE lec7
Type

• Defines the schema of a collection
• It lists all \textit{required} fields
• Fields followed by \textit{?} are \textit{optional}
• CLOSED type = no other fields allowed
• OPEN type = other fields allowed
USE lec7;
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 lec7;
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", "phone": "123456789"}
Types with Nested Collections

USE lec7;
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 lec7;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name : string,
    email: string?
}

USE lec7;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;

{“Name”: “Alice”}
{“Name”: “Bob”}
…
USE lec7;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    myKey: uuid,
    Name : string,
    email: string?
}

USE lec7;
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
Indexes

• Can declare an index on **an attribute of a top-most collection**
  – Cannot index inside nested 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 and NGRAM for textual (string) data.
SELECT ... FROM ... WHERE ... [GROUP BY ...]
SELECT x.mondial FROM world x;

Answer

{“mondial”:
 {“country”: [ country1, country2, …],
  “continent”: […],
  “organization”: […],
 ... 
 ...
}

{“mondial”:
 {“country”: [ country1, country2, …],
  “continent”: […],
  “organization”: […],
 ...
 ...
}
Retrieve countries

{“mondial”: 
{“country”: [ country1, country2, …],
 “continent”: […],
 “organization”: […],
 ...
 ...
 }

Answer

SELECT x.mondial.country FROM world x;

{“country”: [ country1, country2, …],
Retrieve countries, one by one

Answer

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

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

{x: {A: a1, B: [{C: c1, D: d1}, {C: c2, D: d2}]}
{x: {A: a2, B: [{C: c3, D: d3}]}}
{x: {A: a3, B: [{C: c4, D: d4}, {C: c5, D: d5}]}}

xB is a collection
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;
```

```
{"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

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

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';
```

Runtime error:
- 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
{“mondial”: 
  {“country”: [ country1, country2, …],
   “continent”: […],
   “organization”: […],
   …
  }
```

Just the arrays

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

...
Heterogeneous Collections

```sql
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';
```

The problem:

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

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
- 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 x
FROM arr x

SELECT y
FROM arr x, x y

Answers:

{ "x": [ "a", "b" ] }
{ "x": [ ] }
{ "x": [ "b", "c", "d" ] }

Answer:

{ "y": "a" }
{ "y": "b" }
{ "y": "b" }
{ "y": "c" }
{ "y": "d" }
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

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

\[
Unnest_F(coll) = 
\begin{align*}
\{ & A:a1, B:b1, G:\{C:c1\}, \\
   & A:a1, B:b2, G:\{C:c1\}, \\
   & A:a2, B:b3, G[:], \\
   & A:a2, B:b4, G[:], \\
   & A:a2, B:b5, G[:], \\
   & A:a3, B:b6, G:\{C:c2\}, C:c3]\}
\end{align*}
\]

Nested Relational Algebra
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

Nested Relational Algebra

SQL++

DATA514 - Winter 2018
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}]},
   {A:a2, F:[{B:b3}], G:[{C:c1}]},
   {A:a2, F:[{B:b4}], G:[{C:c1}]},
   {A:a2, F:[{B:b5}], G:[{C:c1}]},
   {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]},
   {A:a3, F:[{B:b6}], G:[{C:c2}]},
   {A:a3, F:[{B:b6}], G:[{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}]}]

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
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}]}]

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

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

Nested Relational Algebra

SQL++
Nesting (like group-by)

A flat collection

coll =
\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}
Nesting (like group-by)

A flat collection

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

Nest\(_A\)(coll) =
\[
\{\langle A:a1, \text{GRP:}[[B:b1],[B:b2]] \rangle \}
\{\langle A:a2, \text{GRP:}[[B:b2]] \rangle \}
\]
Nesting (like group-by)

A flat collection

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

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

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

A flat collection

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

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

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

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

\text{FROM coll} \ x
Nesting (like group-by)

A flat collection

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

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

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

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

\[
\text{SELECT DISTINCT x.A, g as GRP FROM coll x}
\]

LET \( g = \) (SELECT y.B FROM coll y WHERE x.A = y.A)
A nested collection

```latex
\text{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
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}]]}

SELECT coll_count(x.F) as cnt
FROM coll x

Answer: { "cnt": 2 } { "cnt": 3 } { "cnt": 1 }

SELECT count(*) as cnt
FROM coll x, x.F y;

Answer: { "cnt": 6 }
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

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

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

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" }
Behind the Scences

Query Processing on NFNF data:

• Option 1: give up on query plans, using 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}]}

DATA514 - Winter 2018
Flattening SQL++ Queries

A nested collection

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

Flat Representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b6</td>
<td></td>
</tr>
</tbody>
</table>
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}]}]

Flat 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>

SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'

SQL++

SQL
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}]}]
```

Flat 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>

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

```
SELECT x.A, y.B
FROM coll x, F 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}]}]
```

Flat 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 x, x.F y
WHERE x.A = 'a1'
```

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, x.F y, x.G 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}]}]

Flat Representation

coll: | F | G
---|---|---
| id | A | parent | B | parent | C
| 1  | a1 | 1     | b1 | 1     | c1
| 2  | a2 | 1     | b2 | 3     | c2
| 3  | a1 | 2     | b3 | 3     | c3

SQL++

SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'

SQL

SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
Semistructured Data Model

- Several file formats: Json, protobuf, XML
- The data model is a tree
- They differ in how they handle structure
```json
{
    "person": [
        {
            "name": "Mary",
            "address": {
                "street": "Maple",
                "no": 345,
                "city": "Seattle"
            }
        },
        {
            "name": "John",
            "address": "Thailand",
            "phone": 2345678
        }
    ]
}
```
JSON vs. XML

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

```xml
<person>
    <name> Mary </name>
    <address>
        <street> Maple </street>
        <no>345 </no>
        <city> Seattle </city>
    </address>
</person>
<person>
    <name> John </name>
    <address> Thailand </address>
    <phone> 2345678 </phone>
</person>
```
JSon vs. XML

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

```xml
<person>
  <name> Mary </name>
  <address>
    <street> Maple </street>
    <no>345 </no>
    <city> Seattle </city>
  </address>
</person>

<person>
  <name> John </name>
  <address> Thailand </address>
  <phone> 2345678 </phone>
</person>
```
JSon vs. XML

```
{
  "person":
    [ {
      "name": "Mary",
      "address": {
        "street": "Maple",
        "no": 345,
        "city": "Seattle"
      }
    },
    {
      "name": "John",
      "address": "Thailand",
      "phone": 2345678
    }
  ]
}
```
JSON vs. XML

```
{
  "person":
  [
    {
      "name": "Mary",
      "address": {
        "street": "Maple",
        "no": 345,
        "city": "Seattle"
      }
    },
    {
      "name": "John",
      "address": "Thailand",
      "phone": 2345678
    }
  ]
}
```
JSon vs. XML

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

Begin tag

```
<person>
  <name> Mary </name>
  <address>
    <street> Maple </street>
    <no> 345 </no>
    <city> Seattle </city>
  </address>
</person>
```

End tag

```
<person>
  <name> John </name>
  <address> Thailand </address>
  <phone> 2345678 </phone>
</person>
```
XML Attributes

Yet another way to represent data

```
<person id = "23">
  <name> Mary </name>
  <address> Seattle </address>
</person>

<person id = "55">
  <name> John </name>
  <address> Thailand </address>
</person>
```
Document Type Definitions (DTD)

- An XML document may have a DTD
- XML document:
  - **Well-formed** = if tags are correctly closed
  - **Valid** = if it has a DTD and conforms to it
- Validation is useful in data exchange

- Use http://validator.w3.org/check to validate
- Superseded by XML Schema (Book Sec. 11.4)
- Very complex: DTDs still used widely
Example DTD

<!DOCTYPE company [
  <!ELEMENT company ((person|product)*)>
  <!ELEMENT person (ssn, name, office, phone?)>
  <!ELEMENT ssn (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT office (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT product (pid, name, description?)>
  <!ELEMENT pid (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
]>
Example DTD

Example of valid XML document:

```xml
<company>
  <person>
    <ssn>123456789</ssn>
    <name>John</name>
    <office>B432</office>
    <phone>1234</phone>
  </person>
  <person>
    <ssn>987654321</ssn>
    <name>Jim</name>
    <office>B123</office>
  </person>
  <product>...</product>
  ...
</company>
```
DTD: The Content Model

• Content model:
  – Complex = a regular expression over other elements
  – Text-only = #PCDATA
  – Empty = EMPTY
  – Any = ANY
  – Mixed content = (#PCDATA | A | B | C)*
<div class="slide">

**DTD: Complex Content**

- **Sequence**
  
  ```
  <!ELEMENT name (firstName, lastName)>
  ```

- **Optional**
  
  ```
  <!ELEMENT name (firstName?, lastName)>
  ```

- **Kleene star**
  
  ```
  <!ELEMENT person (name, phone*)>
  ```

- **Alternation**
  
  ```
  <!ELEMENT person (name, (phone|email))>
  ```

**DTD**

**XML**

```
<name>
  <firstName> . . . . . </firstName>
  <lastName> . . . . . </lastName>
</name>
```

```
<person>
  <name> . . . . . </name>
  <phone> . . . . . </phone>
  <phone> . . . . . </phone>
  . . . . .
</person>
```
Querying XML Data

- XPath = simple navigation

- XQuery = the SQL of XML

- XSLT = recursive traversal
Conclusion

• Semistructured data best suited for data exchange

• For quick, ad-hoc data analysis, use a native query language: SQL++, or AQL, or XQuery
  – Modern, advanced query processors like AsterixDB / SQL++ can process semistructured data as efficiently as RDBMS

• For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS