Introduction to Data Management
CSE 344

Unit 3: NoSQL, JSON, Semistructured Data
(3 lectures*)

*Slides may change: refresh each lecture
Introduction to Data Management
CSE 344

Lecture 11: NoSQL
Announcements

• HW3 (Azure) due tonight (Friday)
• WQ5 (Datalog) due tonight (Friday)
• HW4 (Datalog/Souffle) is posted, due next Friday
• M/W: guest lecturer (Jonathan)
• No class on Friday
• Midterm review on Saturday, 2pm
• Midterm following Monday.
Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
  - NoSQL
  - JSON
  - SQL++
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)
Two Classes of Database Applications

• OLTP (Online Transaction Processing)
  – Queries are simple lookups: 0 or 1 join
    E.g., find customer by ID and their orders
  – Many updates. E.g., insert order, update payment
  – Consistency is critical: transactions (more later)

• OLAP (Online Analytical Processing)
  – aka “Decision Support”
  – Queries have many joins, and group-by’s
    E.g., sum revenues by store, product, clerk, date
  – No updates
NoSQL Motivation

• Originally motivated by Web 2.0 applications
  – E.g. Facebook, Amazon, Instagram, etc
  – Startups need to scale up from 10 to $10^7$ quickly

• Needed: very large scale OLTP workloads
• Give up on consistency, give up OLAP
• NoSQL: reduce functionality
  – Simpler data model
  – Very restricted updates
SQLite:
- One data file
- One user
- One DBMS application

- **Consistency** is easy
- But only a limited number of scenarios work with such model
RDBMS Review: Client-Server

- One server running the database
- Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol
Many users and apps **Consistency** is harder → transactions

- One server running the database
- Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol
Client-Server

• One server that runs the DBMS (or RDBMS):
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)
Client-Server

• **One server that runs the DBMS (or RDBMS):**
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)

• **Many clients run apps and connect to DBMS**
  – Microsoft’s Management Studio (for SQL Server), or
  – psql (for postgres)
  – Some Java program (HW8) or some C++ program
Client-Server

• One server that runs the DBMS (or RDBMS):
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)

• Many clients run apps and connect to DBMS
  – Microsoft’s Management Studio (for SQL Server), or
  – psql (for postgres)
  – Some Java program (HW8) or some C++ program

• Clients “talk” to server using JDBC/ODBC protocol
Web Apps: 3 Tier

Browser

File 1
File 2
File 3
DB Server
Web Apps: 3 Tier

Browser

Connection (e.g., JDBC)

HTTP/SSL

App+Web Server

DB Server

File 1

File 2

File 3
Web Apps: 3 Tier

Web-based applications

File 1

File 2

File 3

DB Server

Connection (e.g., JDBC)

App+Web Server

Browser

HTTP/SSL

CSE 344 - 2019wi
Web Apps: 3 Tier

Web-based applications

File 1
File 2
File 3

DB Server

Connection (e.g., JDBC)

App+Web Server

HTTP/SSL

CSE 344 - 2019wi
Web Apps: 3 Tier

Web-based applications

Replicate App server for scaleup

Why not replicate DB server?

DB Server

File 1

File 2

File 3

App+Web Server

Connection (e.g., JDBC)

HTTP/SSL
Web-based applications

- File 1
- File 2
- File 3

DB Server

Replicate App server for scaleup

Why not replicate DB server? Consistency!

Connection (e.g., JDBC)

HTTP/SSL

App+Web Server
Replicating the Database

• Two basic approaches:
  – Scale up through partitioning – “sharding”
  – Scale up through replication

• Consistency is much harder to enforce
Scale Through Partitioning

• Partition the database across many machines in a cluster
  – Database now fits in main memory
  – Queries spread across these machines

• Can increase throughput

• Easy for writes but reads become expensive!

Application updates here

Three partitions

May also update here
Scale Through Replication

- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become expensive!

App 1 updates here only
Three replicas
App 2 updates here only
Relational Model $\rightarrow$ NoSQL

- Relational DB: difficult to replicate/partition. Eg Supplier(sno,...), Part(pno,...), Supply(sno,pno)
  - Partition: we may be forced to join across servers
  - Replication: local copy has inconsistent versions
  - Consistency is hard in both cases (why?)

- NoSQL: simplified data model
  - Given up on functionality
  - Application must now handle joins and consistency
Data Models

Taxonomy based on data models:

- **Key-value stores**
  - e.g., Project Voldemort, Memcached
- **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB
- **Extensible Record Stores**
  - e.g., HBase, Cassandra, PNUTS
Key-Value Stores Features

• **Data model**: (key, value) pairs
  – Key = string/integer, unique for the entire data
  – Value = can be anything (very complex object)
Key-Value Stores Features

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)

- **Operations**
  - get(key), put(key, value)
  - Operations on value not supported
Key-Value Stores Features

• **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)

• **Operations**
  - get(key), put(key, value)
  - Operations on value not supported

• **Distribution / Partitioning** – w/ hash function
  - No replication: key k is stored at server h(k)
  - 3-way replication: key k stored at h1(k), h2(k), h3(k)
Key-Value Stores Features

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)
- **Operations**
  - get(key), put(key, value)
  - Operations on value not supported
- **Distribution / Partitioning** – w/ hash function
  - No replication: key k is stored at server h(k)
  - 3-way replication: key k stored at h1(k), h2(k), h3(k)

How does get(k) work? How does put(k,v) work?
Example

- How would you represent the Flights data as key, value pairs?
Example

• How would you represent the Flights data as key, value pairs?

• Option 1: key=fid, value=entire flight record
Example

• How would you represent the Flights data as key, value pairs?

• Option 1: key=fid, value=entire flight record

• Option 2: key=date, value=all flights that day
Example

• How would you represent the Flights data as key, value pairs?

• Option 1: key=fid, value=entire flight record

• Option 2: key=date, value=all flights that day

• Option 3: key=(origin,dest), value=all flights between

How does query processing work?
Key-Value Stores Internals

- **Partitioning:**
  - Use a hash function $h$
  - Store every (key, value) pair on server $h(key)$

- **Replication:**
  - Store each key on (say) three servers
  - On update, propagate change to the other servers; *eventual consistency*
  - Issue: when an app reads one replica, it may be stale

- **Usually:** combine partitioning+replication
Data Models

Taxonomy based on data models:

• **Key-value stores**
  - e.g., Project Voldemort, Memcached

• **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB

• **Extensible Record Stores**
  - e.g., HBase, Cassandra, PNUTS
Motivation

• In Key, Value stores, the Value is often a very complex object
  – Key = ‘2010/7/1’, Value = [all flights that date]

• Better: value to be structured data
  – JSON or Protobuf or XML
  – Called a “document” but it’s just data

We will discuss JSON
Data Models

Taxonomy based on data models:

- **Key-value stores**
  - e.g., Project Voldemort, Memcached
- **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB
- **Extensible Record Stores**
  - e.g., HBase, Cassandra, PNUTS
Extensible Record Stores

• Based on Google’s BigTable
• HBase is an open source implementation of BigTable

• Data model:
  – Variant 1: key = rowID, value = record
  – Variant 2: key = (rowID, columnID), value = field

• Will not discuss in class
Introduction to Data Management
CSE 344

Lecture 12:
JSON, Semistructured Data, SQL++
Announcements

- WQ4 (Relational Algebra): due tomorrow
- HW4 (datalog): due next Tuesday

- Midterm: next Wednesday, in class
- Material: up to datalog (Units 1 and 2)

- Review session: Sunday 2pm, GWN 201
Where We Are

- So far we have studied the *relational data model*
  - Data is stored in tables (=relations)
  - Queries are expressions in SQL, relational algebra, or Datalog

- Today: Semistructured data model
  - Popular formats today: XML, JSON, protobuf
JSON - Overview

- JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.

- The filename extension is .json.

We will emphasize JSON as semi-structured data
JSON Syntax

```json
```
JSON vs Relational

• Relational data model
  – Rigid flat structure (tables)
  – Schema must be fixed in advanced
  – Binary representation: good for performance, bad for exchange
  – Query language based on Relational Calculus

• Semistructured data model / JSON
  – Flexible, nested structure (trees)
  – Does not require predefined schema ("self-describing")
  – Text representation: good for exchange, bad for performance
  – Most common use: Language API; query languages emerging
JSON Types

- **Primitive**: number, string, Boolean, null

- **Object**: collection of name-value pairs:
  - `{“name1”: value1, “name2”: value2, ...}
  - “name” is also called a “key”

- **Array**: *ordered* list of values:
  - `[obj1, obj2, obj3, ...]`
Avoid Using Duplicate Keys

The standard allows them, but many implementations don’t

Use an ordered list instead

```json
{id: "07",
 "title": "Databases",
 "author": ["Garcia-Molina", "Ullman", "Widom"]
}
```
JSON Semantics: a Tree!

```
{"person": [
  {
    "name": "Mary",
    "address": {
      "street": "Maple",
      "no": 345,
      "city": "Seattle"
    }
  },
  {
    "name": "John",
    "address": "Thailand",
    "phone": 2345678
  }
]
```
JSON Semantics: a Tree!

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

Recall: arrays are ordered in JSON!
Intro to Semi-structured 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
• ⇒ JSON is more flexible
  – Schema can change per tuple
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>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
<td></td>
</tr>
</tbody>
</table>

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

May inline multiple relations based on foreign keys

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

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

<table>
<thead>
<tr>
<th>Orders</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>personName</td>
<td>date</td>
</tr>
<tr>
<td>John</td>
<td>2002</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
</tr>
</tbody>
</table>
Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

<table>
<thead>
<tr>
<th>Person</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>prodName</td>
</tr>
<tr>
<td>John</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Sue</td>
<td>Phone</td>
</tr>
<tr>
<td></td>
<td>Gadget</td>
</tr>
</tbody>
</table>

Options for the JSON file:
- 3 flat relations: Person, Orders, Product
- Person → Orders → Products:
  - Products are duplicated
- Product → Orders → Person:
  - Persons are duplicated
Semi-structured data

- 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>NULL</td>
</tr>
</tbody>
</table>
Semi-structured data

• 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>
Semi-structured data

- Attributes with different types in different objects

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

- Nested collections
- Heterogeneous collections

- These are difficult to represent in the relational model
Discussion: Why Semi-Structured Data?

- Semi-structured data works well as *data exchange formats*
  - i.e., exchanging data between different apps
  - Examples: XML, JSON, Protobuf (protocol buffers)

- Increasingly, systems use them as a data model for databases:
  - SQL Server supports for XML-valued relations
  - CouchBase, MongoDB, Snowflake: JSON
  - Dremel (BigQuery): Protobuf
Query Languages for Semi-Structured 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
  – AsterixDB: SQL++ (based on SQL)
  – MongoDB: has a pattern-based language
  – JSONiq: http://www.jsoniq.org/
• AsterixDB
  – NoSQL database system
  – Developed at UC Irvine
  – Now an Apache project, being incorporated into CouchDB (another NoSQL DB)

• Uses JSON as data model
• Query language: SQL++
  – SQL-like syntax for JSON data
Asterix Data Model (ADM)

Similar to JSON, with differences. File extension is .adm

- **Primitive types:**
  - Boolean, int, string, date, **uuid** (universally unique identifier), …
  - 123e4567-e89b-12d3-a456-426655440000 is a uuid

- **Null** – JSON Keyword
- **Missing** – Field not actually present

- **Derived types:**
  - **Object**
    ```{"Name": "Alice", "age": 40}```
  - (Ordered) **Array**
    ```[1, 3, "Fred", 1, 9]```
  - (Unordered) **Multiset**
    ```{{1, 3, "Fred", 1, 9}}```
ADM Derived Types

• Based on the JSON standard
• Objects:
  – `{“Name”: “Alice”, “age”: 40}`
  – Fields must be distinct:
    `{“Name”: “Alice”, “age”: 40, “age”:50}`
• Ordered arrays:
  – `[1, 3, “Fred”, 2, 9]`
  – Can contain values of different types
• Multisets (aka bags):
  – `{1, 3, “Fred”, 1, 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
Query FROM Array / Multiset

What do these queries return?

```
SELECT x.phone
FROM [{"name": "Alice", "phone": [300, 150]}] AS x;
```

```
SELECT x.phone
FROM {{ {"name": "Alice", "phone": [300, 150]} } } AS x;
```

```
-- error
SELECT x.phone
FROM {"name": "Alice", "phone": [300, 150]} AS x;
```

Can only query from multi-set or array (not object)
null v.s. missing

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

```sql
SELECT x.b FROM ["a":1, "b":2], ["a":3] AS x;
Answer
{ "b": 2
  
}
```

```sql
SELECT x.b
FROM ["a":1, "b":2], ["a":3, "b":null ] AS x;
Answer
{ "b": 2
  { "b": null }
}
```

```sql
SELECT x.b
FROM ["a":1, "b":2], ["a":3, "b":missing ] AS x;
Answer
{ "b": 2
  { }  
}
Finally, a language that we can use!

```
SELECT x.age
FROM Person AS x
WHERE x.age > 21
GROUP BY x.gender
HAVING x.salary > 10000
ORDER BY x.name;
```

is exactly the same as

```
FROM Person AS x
WHERE x.age > 21
GROUP BY x.gender
HAVING x.salary > 10000
SELECT x.age
ORDER BY x.name;
```
AsterixDB Demo
Introduction to Data Management
CSE 344

Lecture 13: SQL++
Time is Creeping
Review – Big Picture

• NoSQL -> Document Store -> JSON
Review – JSON Standard

- JSON tree structured data
  - Keys and values
  - Objects and arrays
Review – AsterixDB and JSON

• AsterixDB uses a JSON-like data structure
  – Multi-set (unordered array)
  – Missing vs null value semantics
    • Missing: { }
    • Null: { "some_attribute": null }
  – SQL++
Outline

• Data Definition Language (DDL)
  – Defines database schema (CREATE, ALTER, ...)
• Data Manipulation Language (DML)
  – Manipulates data in database (SELECT, INSERT, ...)
• Flexible schema implications
  – Indexing
  – Unnesting
  – Built-in functions
SQL++ Overview

• Data Definition Language: create a
  – Type
  – Dataset (like a relation)
  – Dataverse (a collection of datasets)
  – Index
    • For speeding up query execution

• Data Manipulation Language:
  SELECT - FROM - WHERE
Dataverse

A Dataverse is a Database (i.e., collection of tables)

CREATE DATAVERSE myDB
CREATE DATAVERSE myDB IF NOT EXISTS

DROP DATAVERSE myDB
DROP DATAVERSE myDB IF EXISTS

USE myDB
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 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": []}
Types within Types

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

USE myDB;
DROP TYPE ContactType IF EXISTS;
CREATE TYPE ContactType AS CLOSED {
    Method : string,
    Address: string
}

{"Name": "Carol", "contact": [
    {"Method": "phone", "Address": "1234"},
    {"Method": "email", "Address": "carol@uw.edu"}
]}
```
Datasets

- Dataset = relation/table

- 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?
}
```

Multiset of PersonType objects!

```
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?
}

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

Note: no myKey inserted as it is 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 (HW5 mondial.adm)

• Better: multiple rows, reduced number of nested collections
Example from HW5

mondial.adm is totally semi-structured:
{“mondial”: {“country”: [...], “continent”:[...], …, “desert”:[...]} }

<table>
<thead>
<tr>
<th>country</th>
<th>continent</th>
<th>organization</th>
<th>sea</th>
<th>...</th>
<th>mountain</th>
<th>desert</th>
</tr>
</thead>
<tbody>
<tr>
<td>[{“name”:”Albania”,...}, {“name”:”Greece”,...},...]</td>
<td>...</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Nested objects!

Country.admin, sea.admin, mountain.admin are more structured

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>
</tbody>
</table>
Indexes

• A way to access our data (efficiently)
• Can declare an index on an **top-level type attribute**, i.e. the type used by the dataset
• Will discuss how they work later in the quarter (used to speed up queries)
Indexes

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
Indexes

KEYWORD: good for substring search if your dataset contains strings
Indexes

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

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>

Cannot index inside a nested collection

USE myDB;
CREATE INDEX cityname
  ON country(city.name)
  TYPE BTREE;
AsterixDB Data Model Recap
SQL++ Overview

SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
Retrieve Everything

A collection of objects

1. Bind each object in `world` to `x`

2. Return `mondial` for each `x`

Answer

```json
{{
  "mondial":
  {
    "country": [Albania, Greece, ...],
    "continent": [...],
    "organization": [...],
    ...
  }
}}
```
Retrieve Everything

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

Answer

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

```
{“ans”:
  {“country”: [{Albania}, {Greece}, ...],
   “continent”: [...],
   “organization”: [...],
   ...
  }
}
```
Retrieve countries

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

**Answer**

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

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

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

"-car_code" is an illegal field name
Escape using `\"...\"`
Find each country’s GDP

```sql
SELECT x.mondial.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

```
{ "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": "a4" }
```

```
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"} ]}  
{"A": "a4"}  

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

Answer

Form cross product between each x and its x.B

{"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"}  

CSE 344 - 2019wi 93
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;

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"}
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

```json
[{
  "name": "Albania",
  "gdp_total": 4100
},
{
  "name": "Greece",
  "gdp_total": 101700
},...
]```
In General

Needs to be an array or multiset
(i.e., iterable)

SELECT ...
FROM R AS x, S AS y
WHERE x.f1 = y.f2;

These cannot evaluate to an array or dataset!

Need to “unnest” the array
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:

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

Error: Type mismatch!

city is an array

city is an object
Return province and city names

The problem:

```
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);
```
Return province and city names

The problem:

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

```
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);
```

Note: get name directly from z

City is an array

City is an object
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";
```
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";
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
- Splitting
- SQL++ ⇒ SQL
  - Semistructured ⇒ Relational
Basic Unnesting

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

```
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}]}]
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\textsubscript{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

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

Unnest_{F}(\text{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

Refers to relations defined on the left
Unnesting Specific Field

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:\[]\}, \\
\{A:a3, F:\{B:b6\}, G:\{C:c2, C:c3\}\}\}
\]

Unnest\(_F_\)(\text{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\}\}\}

\[
\text{SELECT x.A, y.B, x.G} \\
\text{FROM coll x, x.F y} \\
\]

\[
\text{=} \\
\text{SELECT x.A, y.B, x.G} \\
\text{FROM coll x, UNNEST x.F y}
\]
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
Unnesting Specific Field

A nested collection

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

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

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

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
Nesting

We want:

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

Using LET syntax:

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

```
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)
```
Nesting (like group-by)

A flat collection

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

A flat collection

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

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

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

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

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

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

A flat collection

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

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 F array for each A

SELECT x.A, strict_count(x.F) AS cnt
FROM C AS x

These are NOT equivalent!

SELECT x.A, COUNT(*) AS cnt
FROM C AS x, x.F AS y
GROUP BY x.A
## Grouping and Aggregates

<table>
<thead>
<tr>
<th>Function</th>
<th>NULL</th>
<th>MISSING</th>
<th>Empty Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRICT_COUNT</td>
<td>counted</td>
<td>counted</td>
<td>0</td>
</tr>
<tr>
<td>STRICT_SUM</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>STRICT_MAX</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>STRICT_MIN</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>STRICT_AVG</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_COUNT</td>
<td>not counted</td>
<td>not counted</td>
<td>0</td>
</tr>
<tr>
<td>ARRAY_SUM</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MAX</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MIN</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_AVG</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
</tbody>
</table>
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

SELECT x.A, x.B, y.C
FROM coll1 AS x 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} ]
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 !
Splitting

• Recall: a many-to-one relation should have one foreign key, from “many” to “one”
• Sometimes people represent it in the opposite direction, from “one” to “many”:
  – 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
SELECT y.name, z, x.gdp_total
FROM country AS x, river AS y,
    split(y. `-country`, " ") AS z
WHERE x. `-car_code` = z

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

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

["name": "Donau",   "-country": "SRB A D H HR SK BG AL MD UA"],
{"name": "Colorado", "-country": "MEX USA"},
... ]
Behind the Scenes
i.e., "How to execute SQL++ queries internally?"

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

Is it possible to (1) store nested data in flat relational form and (2) run standard relational queries over it?
Flattening SQL++ Queries

A nested collection

coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:b1}]}],
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
{A:a1, F:[{B:b6}], G:[{C:b2},{C:b3}]}]}
Flattening SQL++ Queries

A nested collection

```
coll = [{A:a1, F:{B:b1}, B:b2}, G:{C:b1}]
{A:a2, F:{B:b3}, B:b4, B:b5}, G:[ ]
{A:a1, F:{B:b6}, G:{C:b2}, C:b3}]
```

Relational representation

<table>
<thead>
<tr>
<th>coll:</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A</td>
<td>parent</td>
</tr>
<tr>
<td>1</td>
<td>a1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b3</td>
<td></td>
</tr>
</tbody>
</table>
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = “a1”
Flattening SQL++ Queries

A nested collection

coll = 

{A:a1, F:[{B:b1},{B:b2}], G:[{C:b1}]}, 
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]}, 
{A:a1, F:[{B:b6}], G:[{C:b2},{C:b3}]}

Relational representation

<table>
<thead>
<tr>
<th>coll:</th>
<th>id</th>
<th>A</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>a1</td>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>a2</td>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>a1</td>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

Answer:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a1</td>
<td>b2</td>
</tr>
<tr>
<td>a1</td>
<td>b6</td>
</tr>
</tbody>
</table>
### Flattening SQL++ Queries

#### A nested collection

```
coll =
[ {A:a1, F:[{B:b1},{B:b2}], G: [{C:b1}]},
  {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[]},
  {A:a1, F:[{B:b6}], G: [{C:b2},{C:b3}]} ]
```

#### Relational representation

**F**

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b3</td>
</tr>
</tbody>
</table>

**G**

<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>b3</td>
</tr>
</tbody>
</table>

#### SQL++

```sql
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = “a1”
```

#### SQL

```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**

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

**Relational representation**

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**SQL++**

```
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

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

Relational representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>F 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>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>b1</td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>b3</td>
</tr>
</tbody>
</table>

SQL++

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

Answer:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
</tbody>
</table>
Flattening SQL++ Queries

A nested collection

```
coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:b1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ }],
 {A:a1, F:[{B:b6}], G:[{C:b2},{C:b3}]}]
```

Relational representation

```
coll:

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>F</th>
<th>G</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>2</td>
<td></td>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>b6</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>b1</td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
</tr>
<tr>
<td>3</td>
<td>b3</td>
</tr>
</tbody>
</table>
```

SQL++

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

SQL

```
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
```

Answer:

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
</tbody>
</table>
```
Semistructured Data Model

- Several file formats: JSON, protobuf, XML
- Data model = Tree
- Differ in how they handle structure:
  - Open or closed
  - Ordered or unordered
- Query language take NFNF into account
  - Various “extra” constructs introduced as a result
    - Nesting & Unnesting, strict aggregates, splitting
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 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