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 on Friday
- HW4 (datalog) due next Friday
- Midterm next Friday (May 3rd)

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)

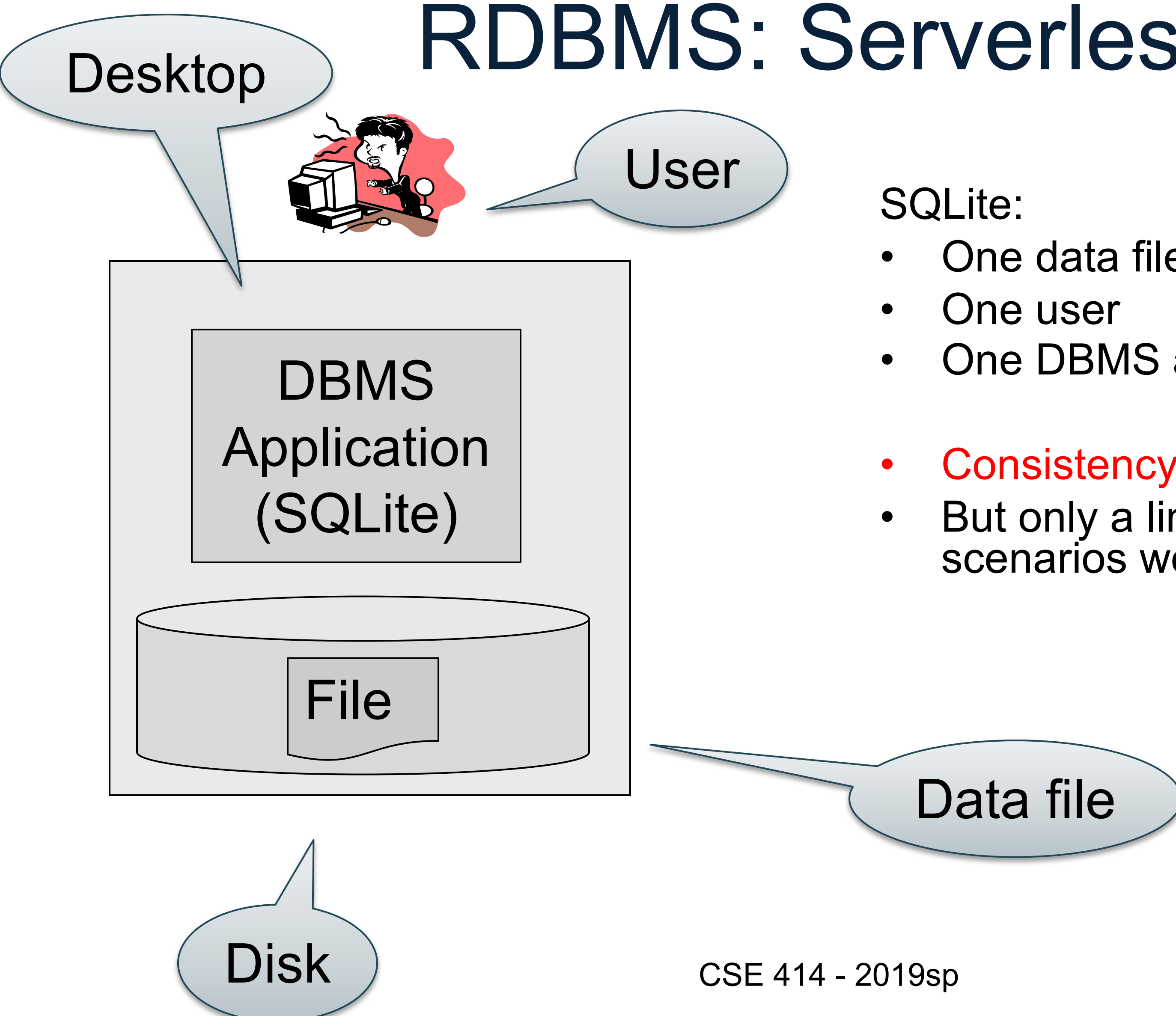
Two Classes of Database Applications

- 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

RDBMS Architectures

- Serverless
- 2 tier: client/server
- 3 tier: client/app-server/db-server

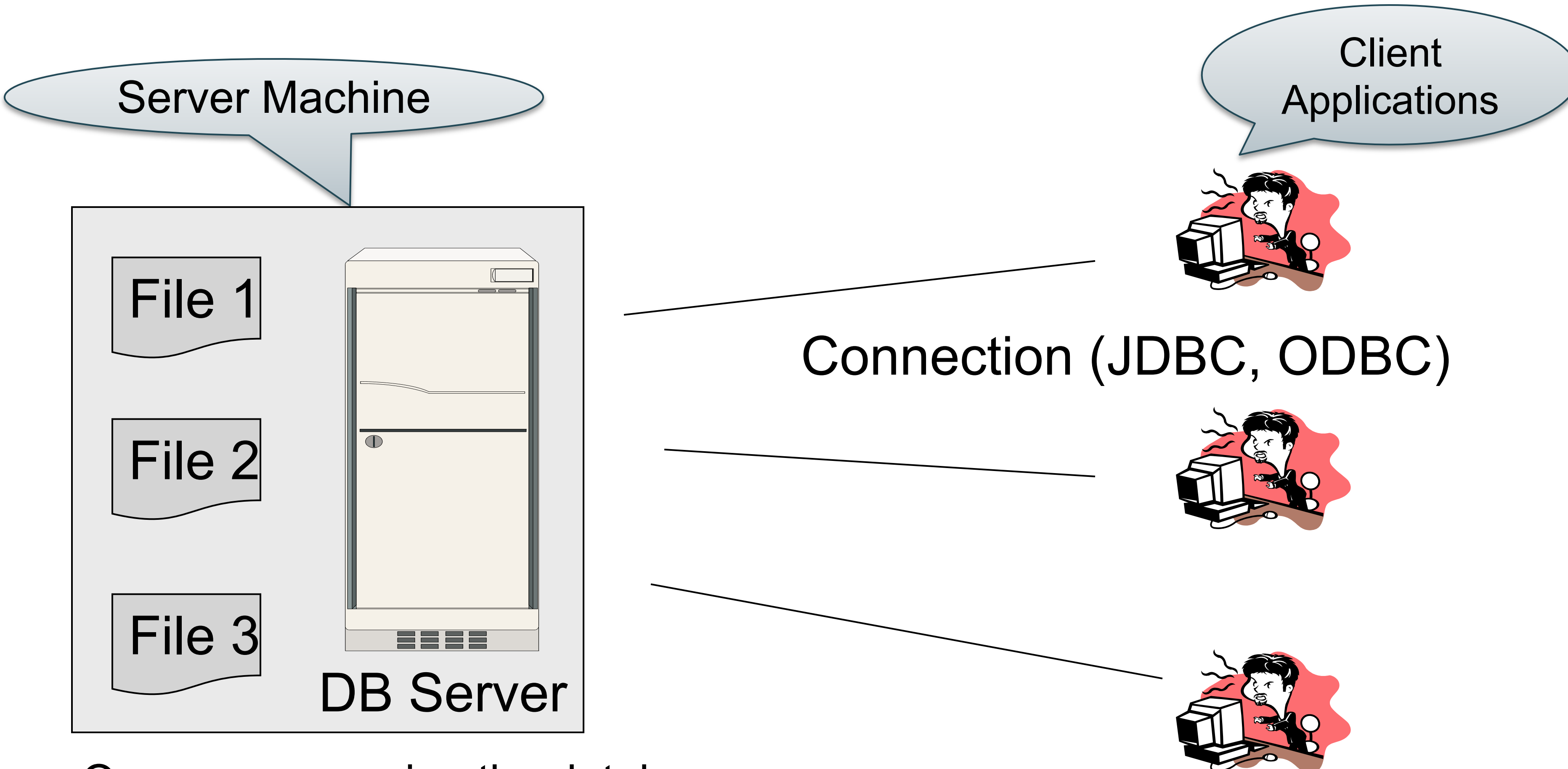
RDBMS: Serverless



SQLite:
- One data file
- One user
- One DBMS application

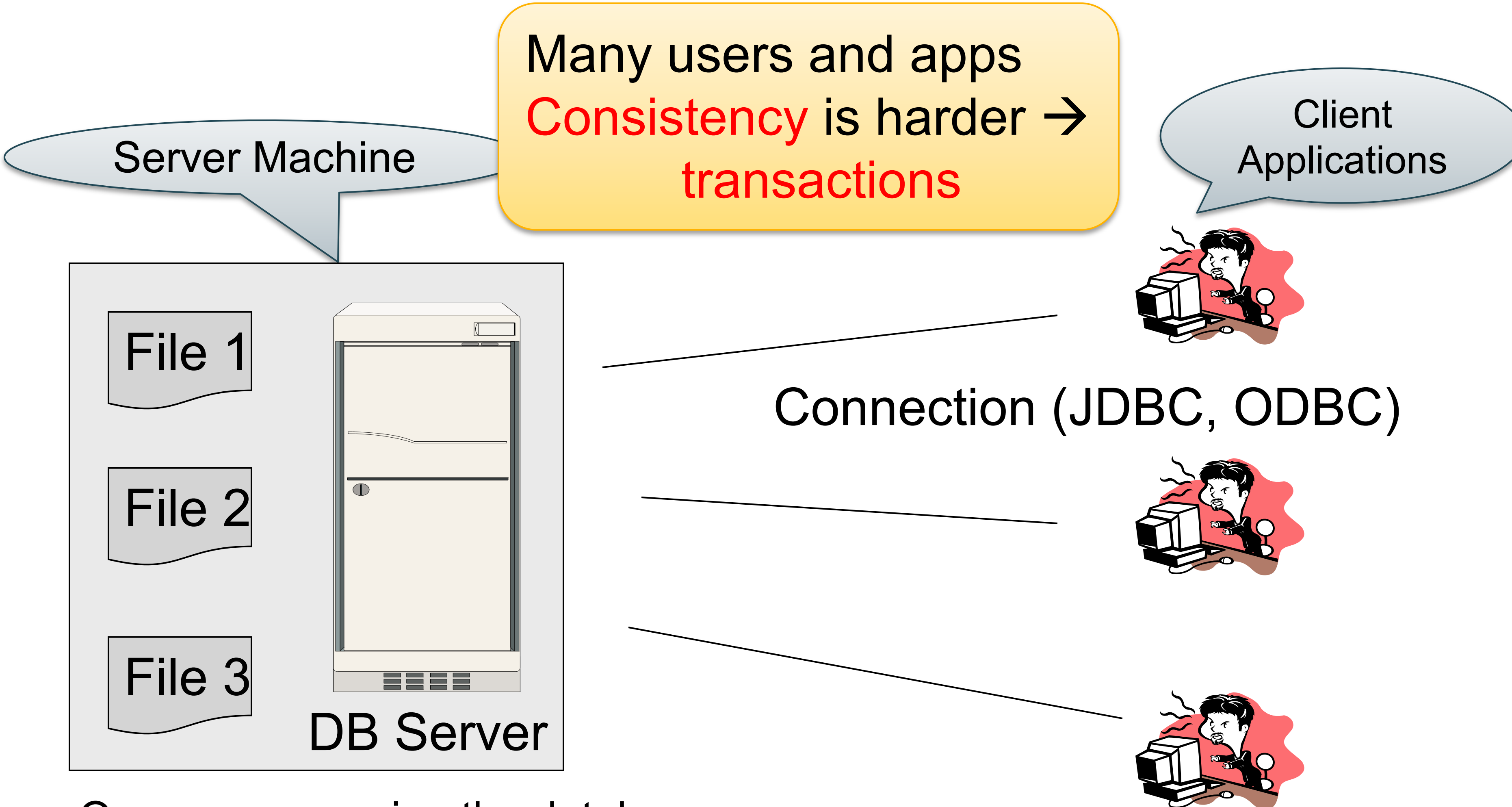
- Consistency is easy
- But only a limited number of scenarios work with such model

RDBMS: Client-Server



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

RDBMS: Client-Server



- 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)

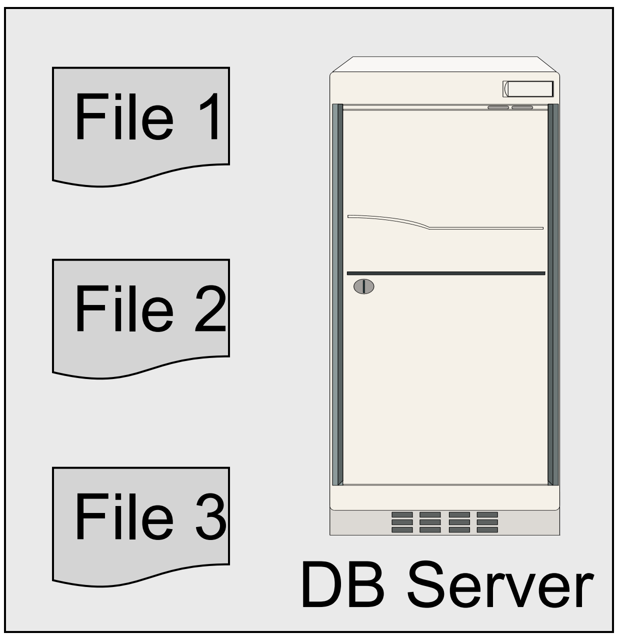
Client-Server

- 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

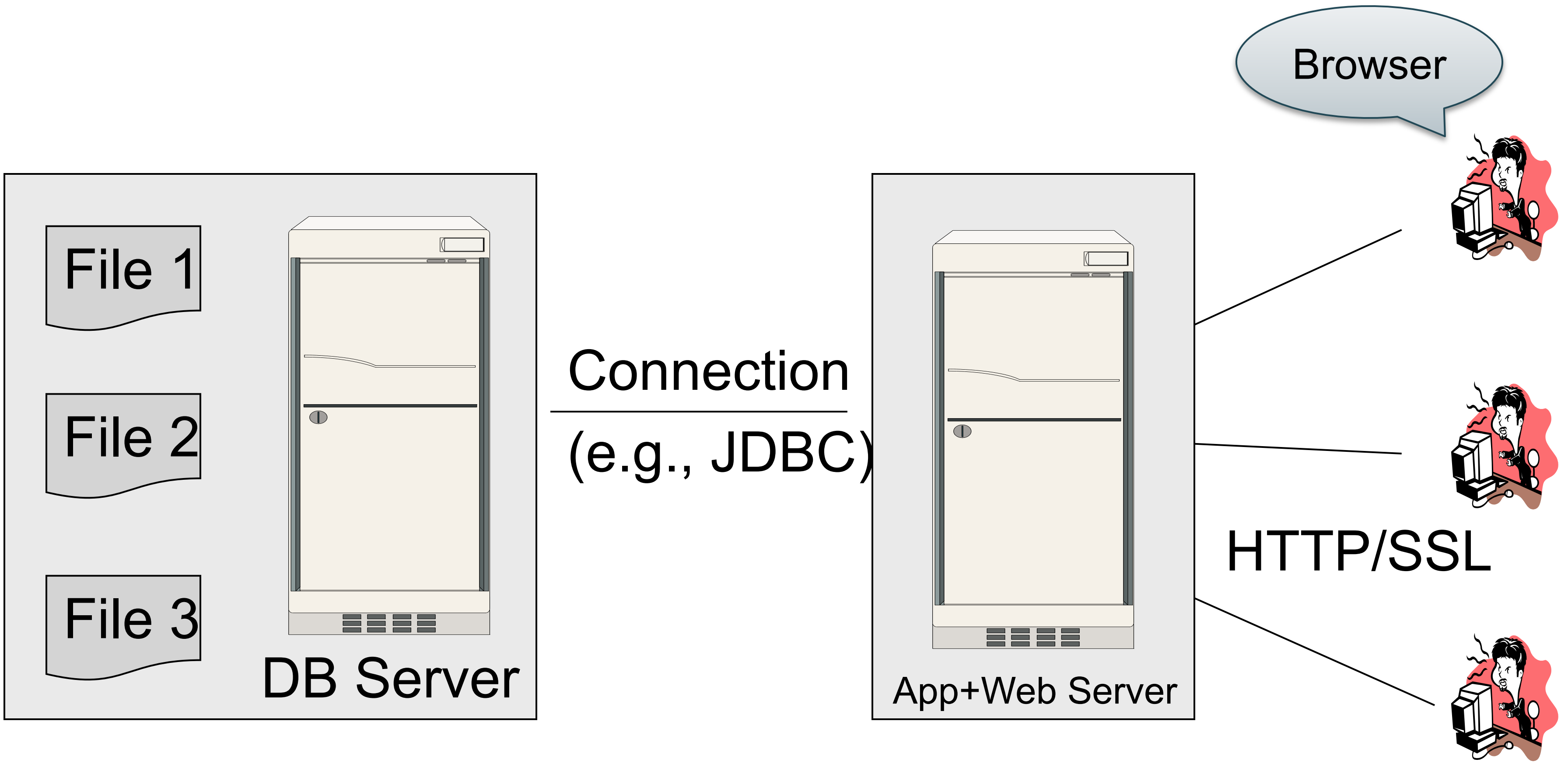
Client-Server

- 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

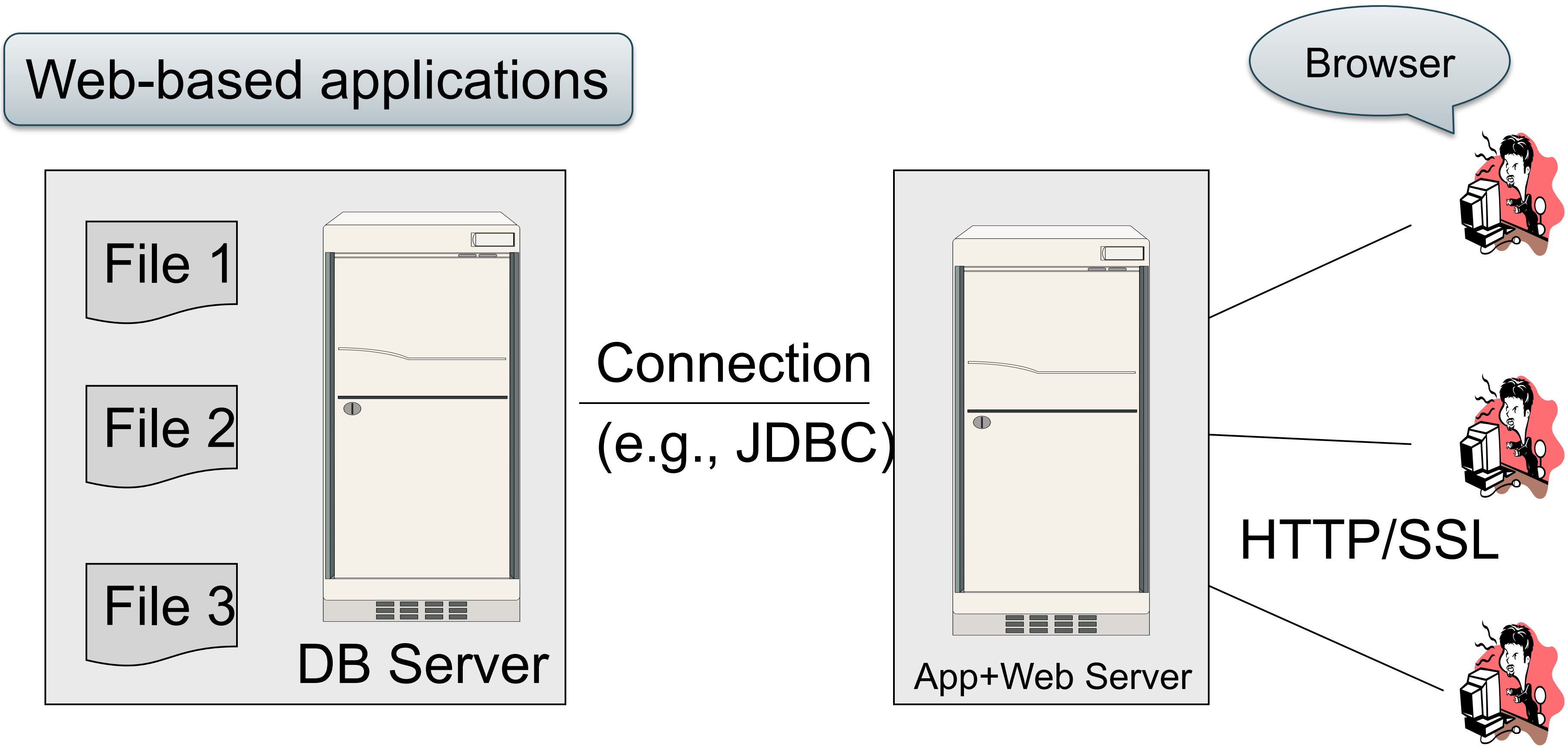
Web Apps: 3 Tier



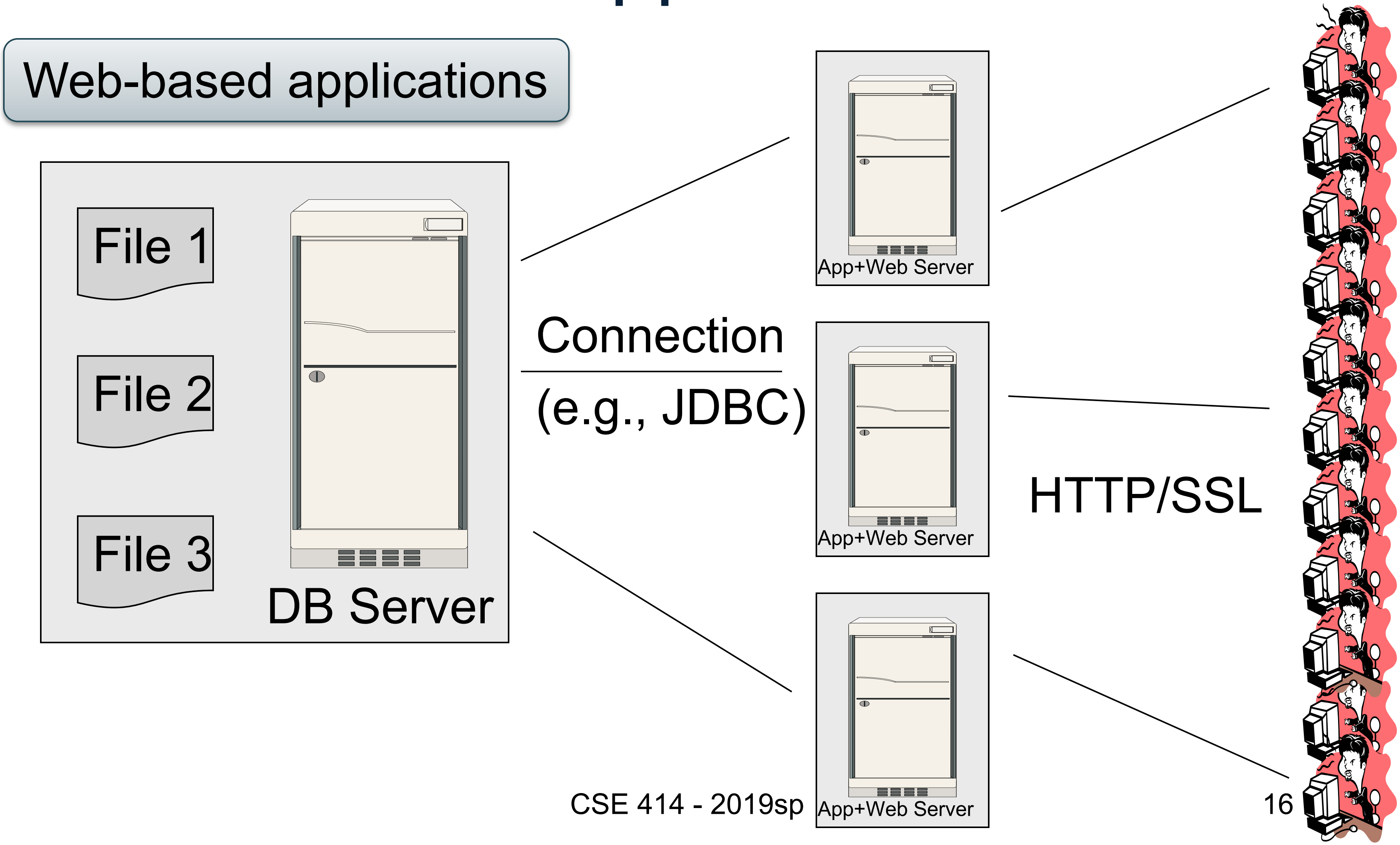
Web Apps: 3 Tier



Web Apps: 3 Tier



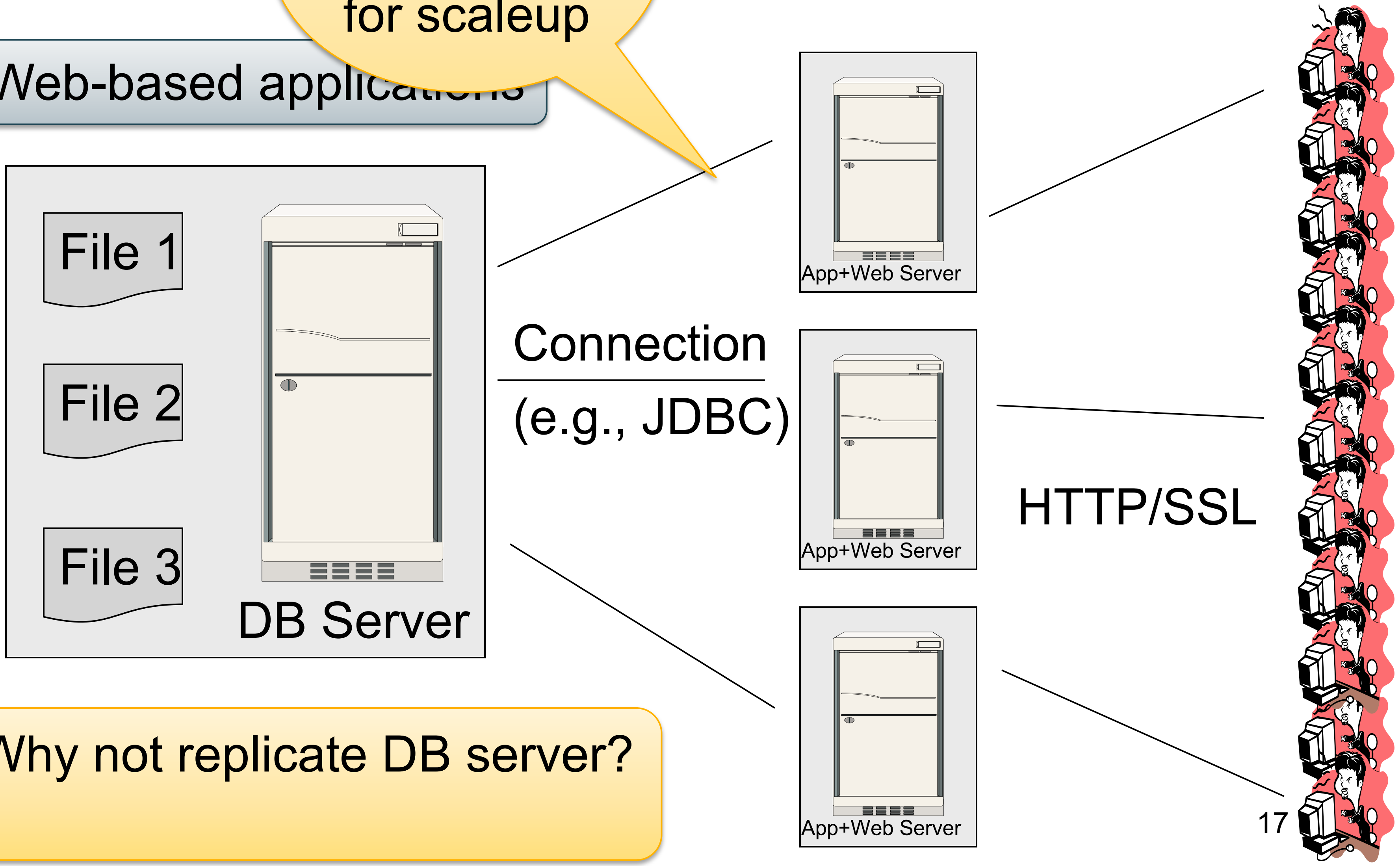
Web Apps: 3 Tier



...ps: 3 Tier

Replicate App server for scaleup

Web-based applications

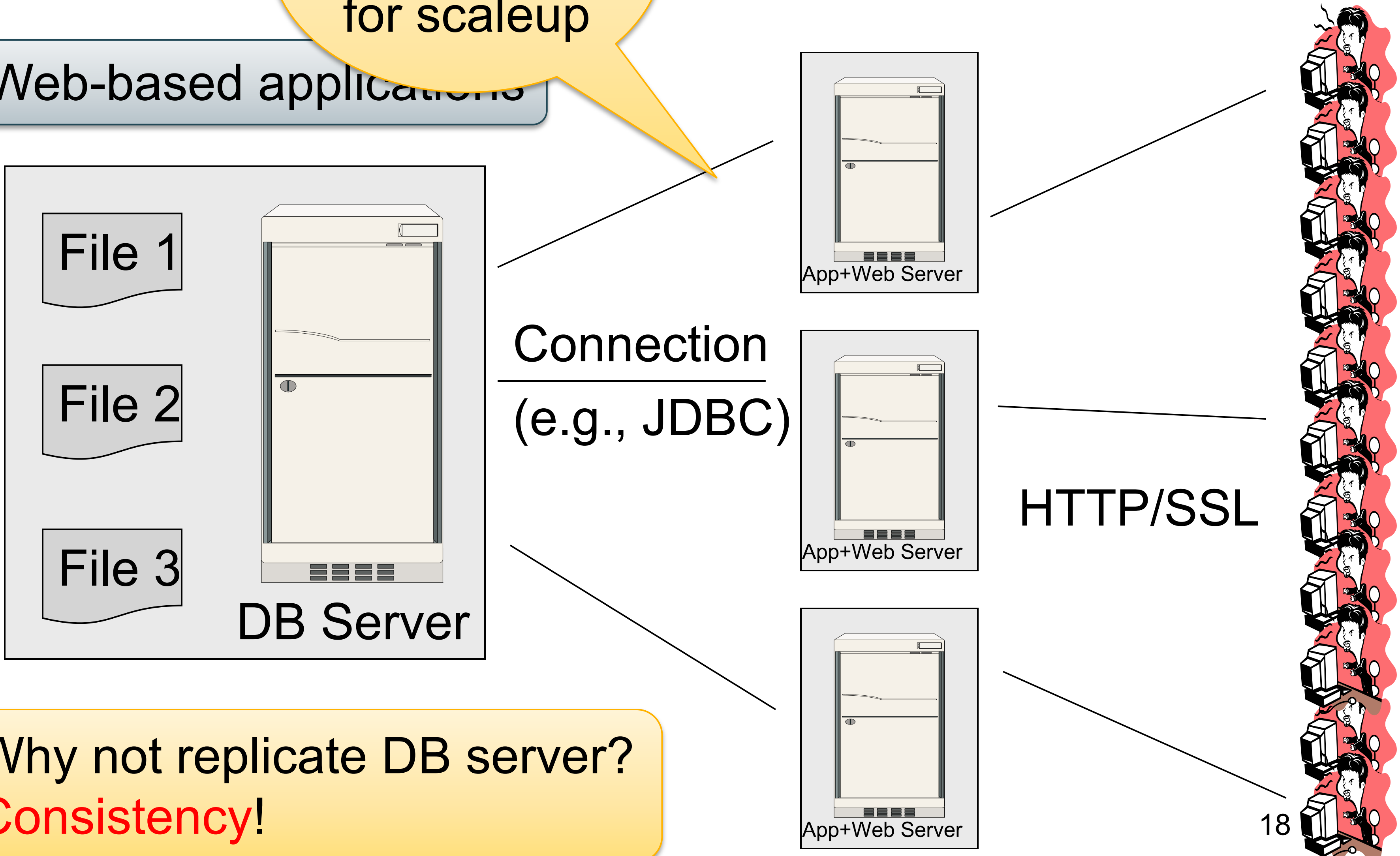


Why not replicate DB server?

Replicate App server for scaleup

ps: 3 Tier

Web-based applications



Why not replicate DB server?
Consistency!

NoSQL Motivation

- Originally motivated by Web 2.0 applications
  - E.g. Facebook, Amazon, Instagram, etc
  - Startups need to scaleup from 10 to 10^7 quickly

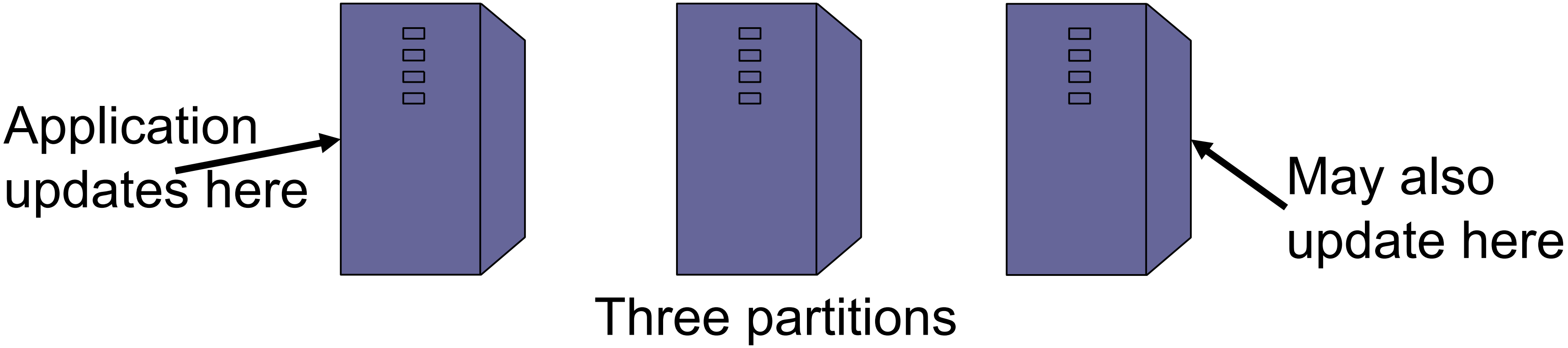
- Needed: very large scale OLTP workloads
- Give up on consistency, give up OLAP
- NoSQL: reduce functionality
  - Simpler data model
  - Very restricted updates

Replicating the Database

- Two basic approaches:
  - Scale up through partitioning – “sharding”
  - Scale up through replication
- Consistency is much harder to enforce

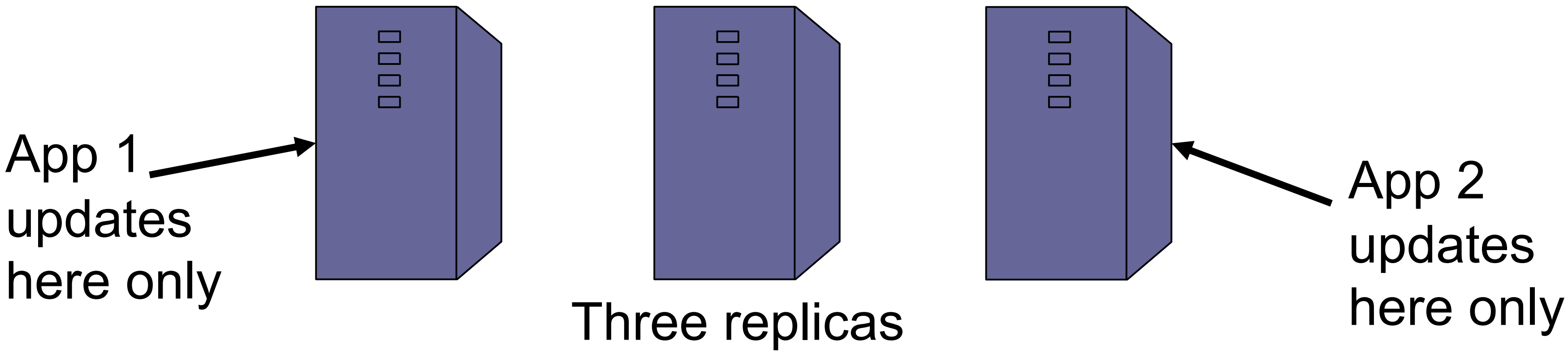
Scale Through Partitioning

- 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!



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!



Relational Model → 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?

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

Example

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

How does query processing work?

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

Example

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

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

How does query processing work?

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

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

How does query processing work?

Flights(fid, date, carrier, flight_num, origin, dest, ...)
Carriers(cid, name)

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++

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

```
{  "book": [
      {"id":"01",
       "language": "Java",
       "author": "H. Javeson",
       "year": 2015
      },
      {"id":"07",
       "language": "C++",
       "edition": "second"
       "author": "E. Sepp",
       "price": 22.25
      }
   ]
}
```

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

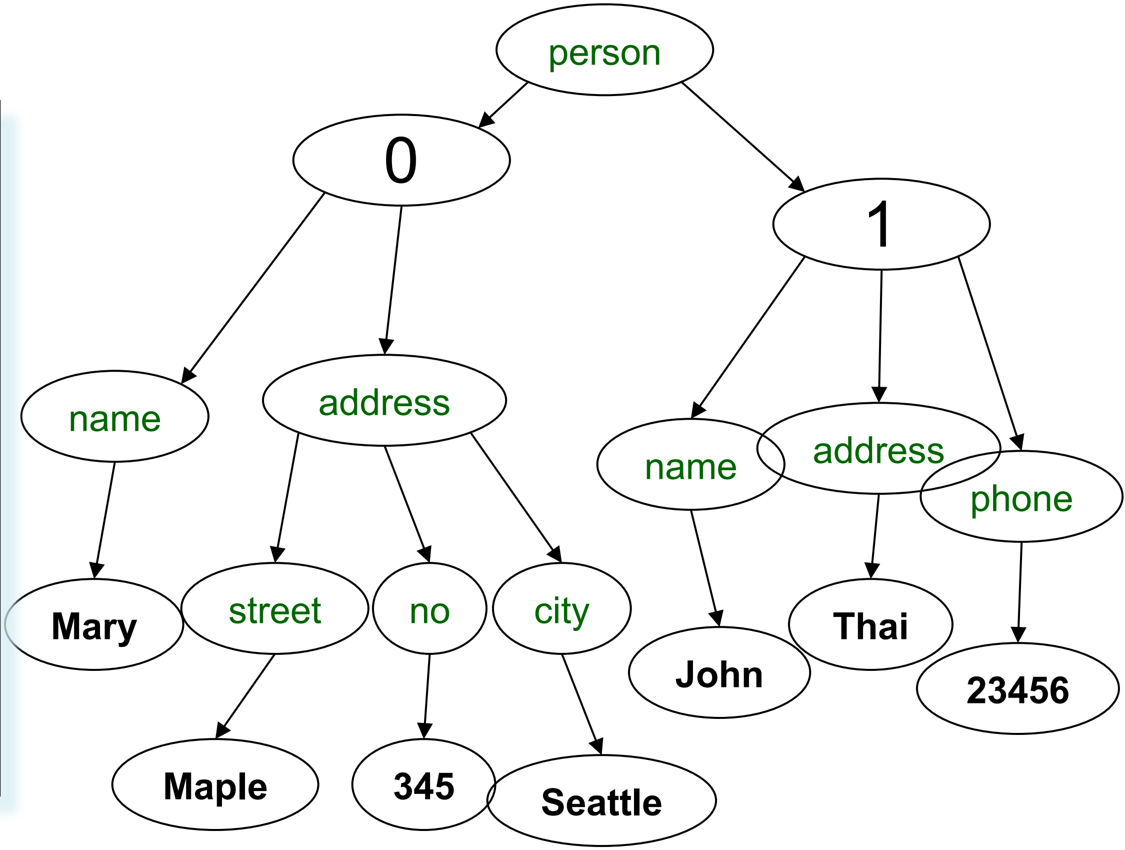
Use an ordered list instead

```
{"id":"07",
    "title": "Databases",
    "author": "Garcia-Molina",
    "author": "Ullman",
    "author": "Widom"
}
```

```
{"id":"07",
    "title": "Databases",
    "author": ["Garcia-Molina",
               "Ullman",
               "Widom"]
}
```

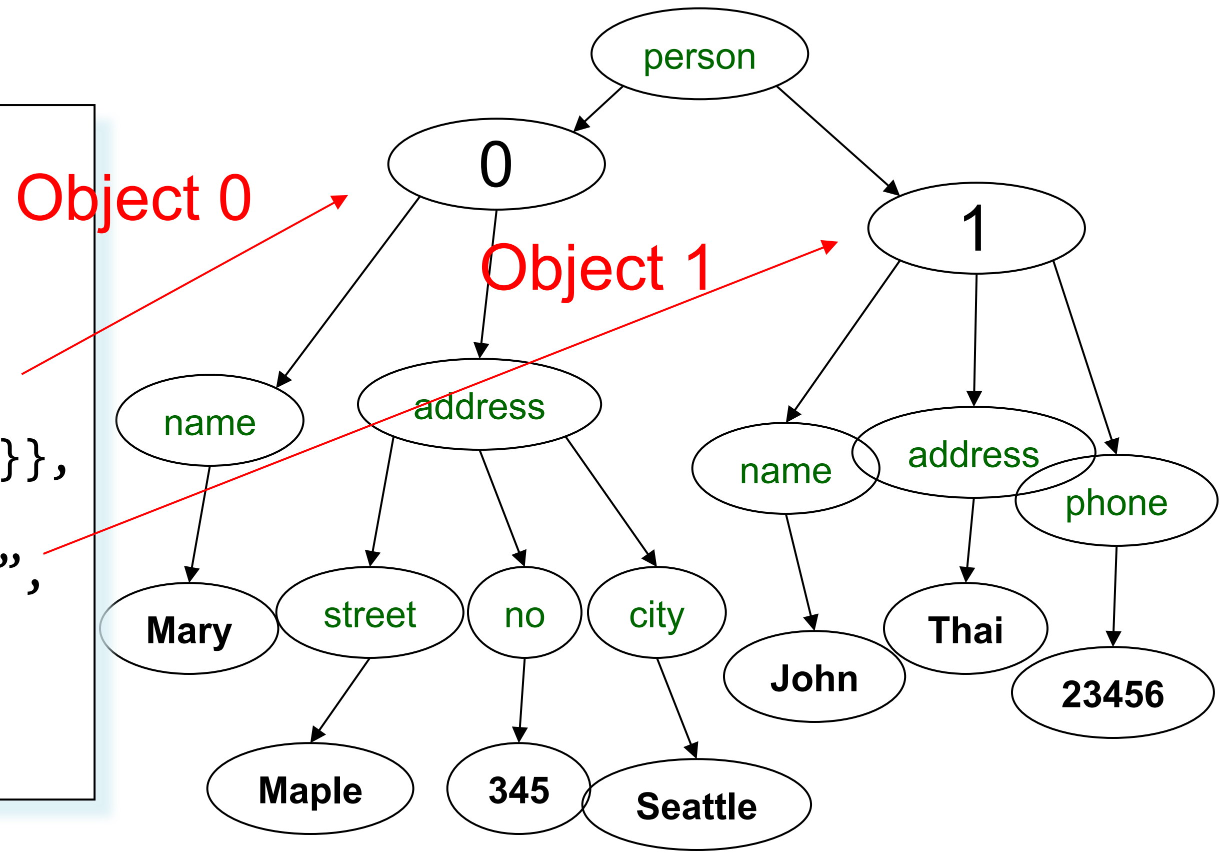
JSON Semantics: a Tree !

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



JSON Semantics: a Tree !

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

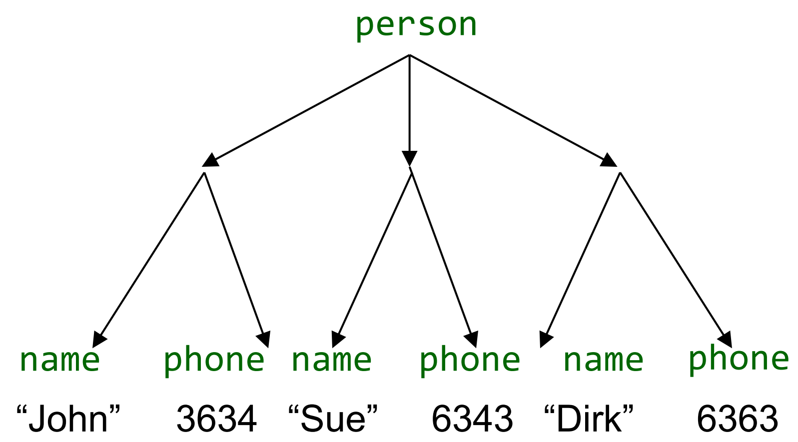


Recall: arrays are *ordered* in JSON!

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



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 multiple relations based on 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"}
   ]
  }
 ]
}
```

Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

Person

name	phone
John	3634
Sue	6343

Product

prodName	price
Gizmo	19.99
Phone	29.99
Gadget	9.99

Orders

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

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:

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

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	NULL

Semi-structured data

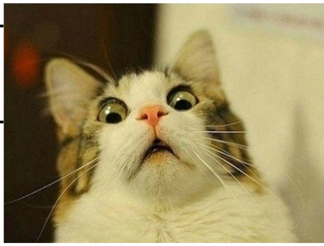
- Repeated attributes

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

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456



Semi-structured data

- 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

- 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:

- Dremel (~ SQL): google internal
- BigQuery (~ SQL): google external

JSON:

- CouchBase: N1QL
- AsterixDB: SQL++ (~ SQL)
- MongoDB: 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

ADM Derived Types

- Based on the JSON standard
- Objects:
  - `{“Name”: “Alice”, “age”: 40}`
  - Fields must be distinct:
    `{“Name”: “Alice”, “age”: 40, ~~“age”:50~~}`

Can’t have repeated fields

- Ordered arrays:
  - `[1, 3, “Fred”, 2, 9]`
  - Can contain values of different types
- Multisets (aka bags):
  - `{{1, 3, “Fred”, 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

Basic Queries

What do these queries return?

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

Answer: {“name”: “Alice”}

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

Answer: {“phone”: [300, 150]}

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


Answer: {“name”: “Alice”, “phone”: [300, 150]}

Query FROM Array / Multiset

What do these queries return?

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

Answer: {“name”: “Alice”}

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

Answer: the same

Can only query from multi-set or array (not object)

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

Query Nested Collections

What do these queries return?

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

Answer
```
300
150
```

```
SELECT y
FROM [{"name": "Alice", "phone": [300, 150]}] AS x,
     x.phone AS y
WHERE y > 200;
```

Answer
```
300
```

Query Semi-structured Data

What do these queries return?

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

Answer
```
{"a": 1}
{"a": 3}
```

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

Answer
```
{"a":1, "b":2}
{"a":3 }
```

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

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

Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, …), date, time, etc
- UUID = universally unique identifier
  Use it as a system-generated unique key
- Values:
  - NULL means null
  - MISSING means it's not there (see next)

`null` v.s. `missing`

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

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

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

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

Answer
```
{"b": 2}
{"b": null  }
```

```
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;
```

FWGHOS lives!!

Introduction to Data Management
CSE 344

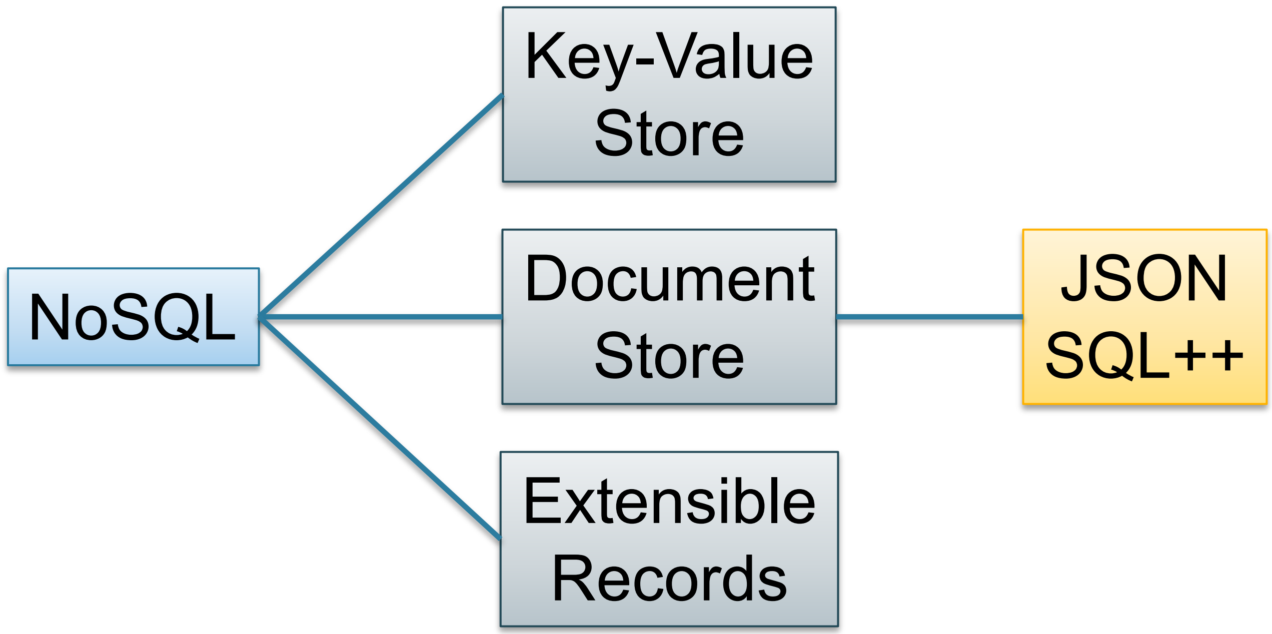
Lecture 13: SQL++

Announcements

- HW3 is due tonight!
- Midterm next Friday
  - Cover material up to date
- HW4 due next Friday

Review – Big Picture

- NoSQL -> Document Store -> JSON



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?
}
```

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

Set 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?
}
```

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

Note: no `myKey` inserted as it is autogenerated

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

JSON 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":[...]}}
```

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

Nested objects!

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

Country:

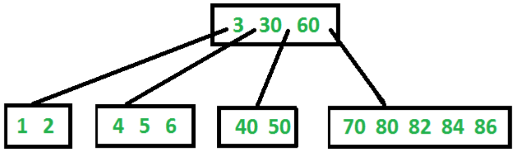
-car_code	name	...	ethnicgroups	religions	...	city
AL	Albania	...	[...]	[...]	...	[...]
GR	Greece	...	[...]	[...]	...	[...]
...	...	...	...			

Indexes

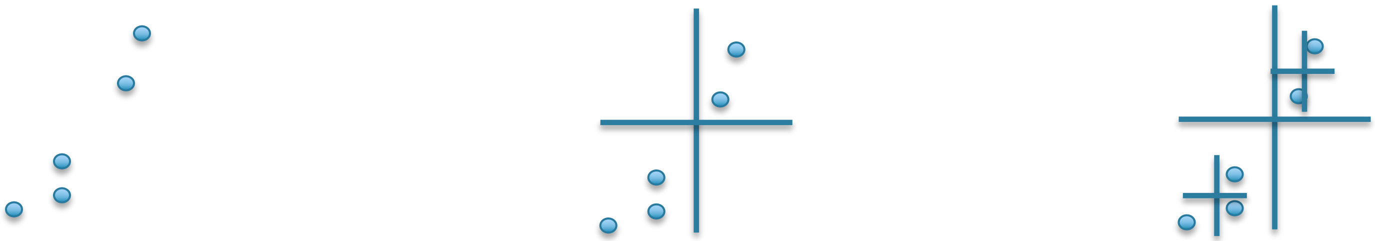
- A way to access our data (efficiently)
- Can declare an index on an **top-level type attribute**, i.e. the type used by the dataset
- Will discuss how they work later in the quarter (used to speed up queries)

Indexes

`BTREE`: good for equality and range queries
E.g., `name="Greece"`; `20 < age and age < 40`

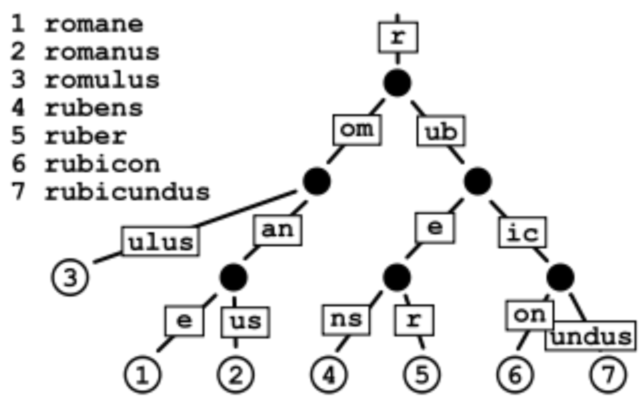


`RTREE`: good for 2-dimensional range queries
E.g., `20 < x and x < 40 and 10 < y and y < 50`



Indexes

KEYWORD: good for substring search if your dataset contains strings

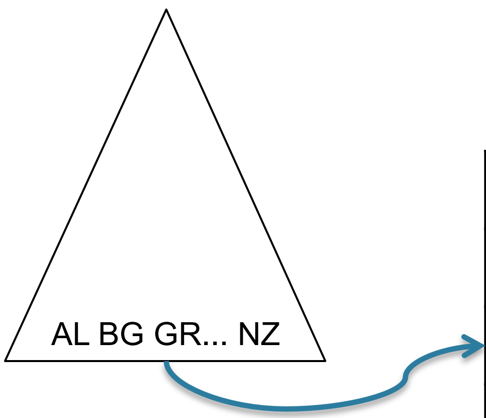


Indexes

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

Cannot index inside a nested collection

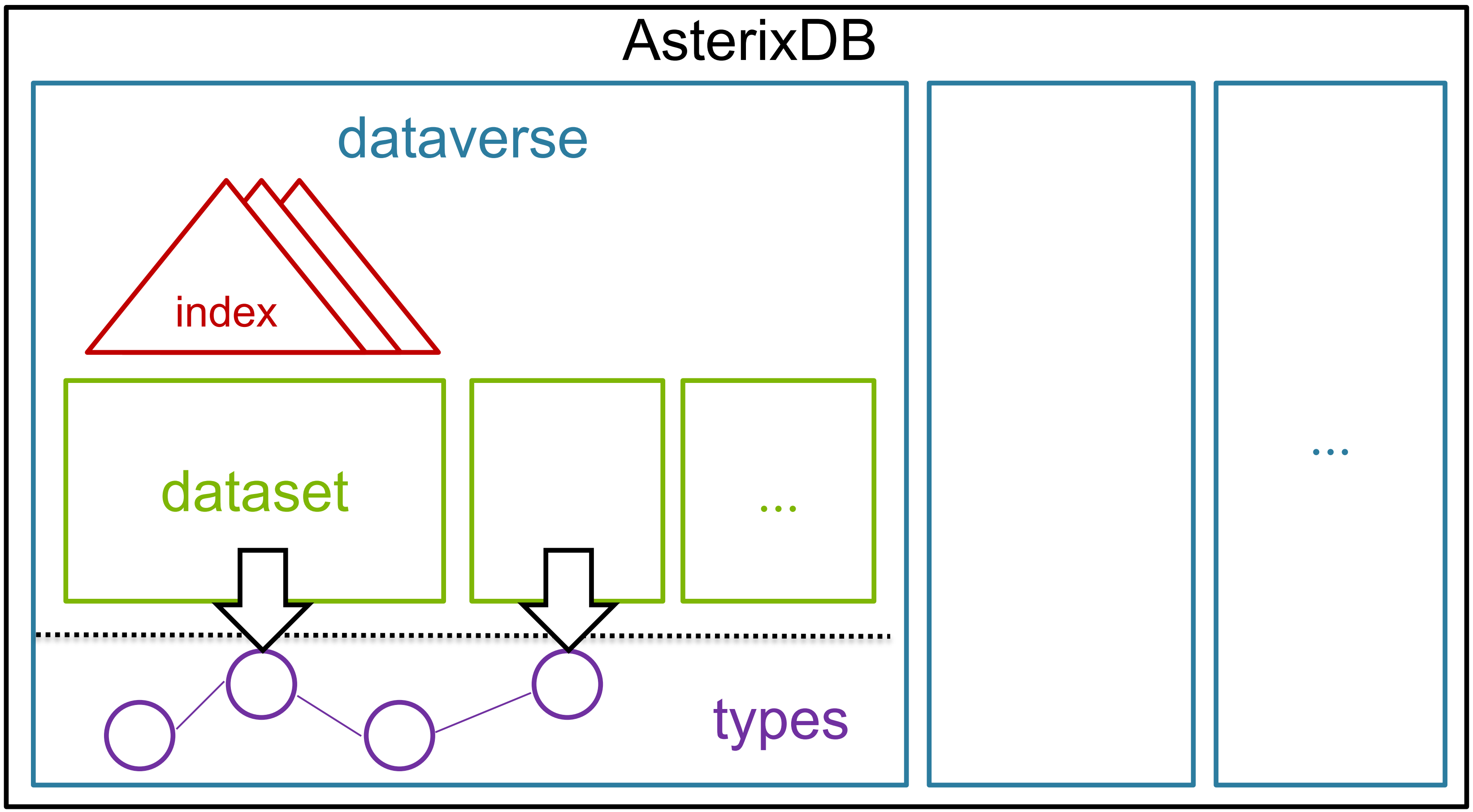
```
USE myDB;
CREATE INDEX cityname
     ON country(city.name)
     TYPE BTREE;
```



Country:

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

AsterixDB Data Model Recap



SQL++ Overview

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

world

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

Retrieve Everything

A collection of objects

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

2. Return mondial for each x

1. Bind each object in world to x

Answer

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

Retrieve Everything

world

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

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

Answer

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

world

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

Retrieve countries

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

Answer

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

Find each country's GDP

world

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

country

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

"-car_code" is an illegal field name
Escape using ` ... `

```
SELECT x.mondial.country.name, c.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,
     ...
   }, ...
}}
```

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`;
```

`x.mondial.country` is an array of objects. No field as `-car_code`!

`Error: Type mismatch!`

Need to "unnest" the array

Unnesting collections

mydata

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

Form cross product between each x and its x.B

Answer

```
{"A": "a1", "C": "c1", "D": "d1"}
{"A": "a1", "C": "c2", "D": "d2"}
{"A": "a2", "C": "c3", "D": "d3"}
{"A": "a3", "C": "c4", "D": "d4"}
{"A": "a3", "C": "c5", "D": "d5"}
```

Unnesting collections

mydata

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

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

Same as before

Answer

```
{"A": "a1", "C": "c1", "D": "d1"}
{"A": "a1", "C": "c2", "D": "d2"}
{"A": "a2", "C": "c3", "D": "d3"}
{"A": "a3", "C": "c4", "D": "d4"}
{"A": "a3", "C": "c5", "D": "d5"}
```

world

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

country

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

Find each country’s GDP

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

Answer

```
{ "name": "Albania", "gdp_total": "4100" }
{ "name": "Greece", "gdp_total": "101700" }
...
```

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

world

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

Return province and city names

(each country may have many provinces and cities)

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name="Greece";
```

The problem:

Error: Type mismatch!

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

city is an array

city is an object

world

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

Return province and city names

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name="Greece" AND IS_ARRAY(z.city);
```

The problem:

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

city is an array

city is an object

world

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

Return province and city names

Note: get name directly from z

```
SELECT z.name AS province_name, z.city AS city_name
FROM world x, x.mondial.country y, y.province z
WHERE y.name="Greece" AND NOT IS_ARRAY(z.city);
```

The problem:

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

city is an array

city is an object

world

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

Return province and city names

```
SELECT z.name AS province_name, u.name AS city_name

FROM world x, x.mondial.country 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";
```

Get both!

world

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

Return province and city names

```
SELECT z.name AS province_name, u.name AS city_name
FROM world x, x.mondial.country y, y.province z,

     (CASE WHEN z.city IS missing THEN []
           WHEN IS_ARRAY(z.city) THEN z.city
           ELSE [z.city] END) AS u

WHERE y.name="Greece";
```

Even better

Useful Functions

- `is_array`
- `is_boolean`
- `is_number`
- `is_object`
- `is_string`
- `is_null`
- `is_missing`
- `is_unknown = is_null` or `is_missing`

Useful Paradigms

- Unnesting
- Nesting
- Grouping and aggregate
- Joins
- Splitting
- SQL++ ⇨ SQL
  - Semistructured ⇨ Relational

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}]}]
```

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}]}]
```

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}]}]
```

Nested Relational Algebra

```
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}]}]
```

$\text{Unnest}_F(\text{coll})$ =

```
[{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}]}]
```

Nested Relational Algebra

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

=

```
SELECT x.A, y.B, x.G
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}]}]

Nested Relational Algebra

$Unnest_G$(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]}

Nested Relational Algebra

```
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
```

Nesting

C

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

We want:

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

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

Using LET syntax:

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

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

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

Nested Relational Algebra

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

Nesting (like group-by)

A flat collection

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

Nested Relational Algebra

$Nest_A$(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}]}]

Nesting (like group-by)

A flat collection

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

Nested Relational Algebra

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
```

Nesting (like group-by)

A flat collection

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

Nested Relational Algebra

$Nest_A$(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
```

```
SELECT x.A, COUNT(*) AS cnt
FROM C AS x, x.F AS y
GROUP BY x.A
```

These are NOT equivalent! (why?)

Grouping and Aggregates

Function	NULL	MISSING	Empty Collection
STRICT_COUNT	counted	counted	0
STRICT_SUM	returns NULL	returns NULL	returns NULL
STRICT_MAX	returns NULL	returns NULL	returns NULL
STRICT_MIN	returns NULL	returns NULL	returns NULL
STRICT_AVG	returns NULL	returns NULL	returns NULL
ARRAY_COUNT	not counted	not counted	0
ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

Joins

Two flat collection

```
coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1, C:c1}, {B:b1, C:c2}, {B:b3, C:c3}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 AS x, coll2 AS y
WHERE x.B = y.B
```

Answer

```
[{A:a1, B:b1, C:c1},
 {A:a1, B:b1, C:c2},
 {A:a2, B:b1, C:c1},
 {A:a2, B:b1, C:c2}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 AS x JOIN coll2 AS y ON x.B = y.B
```

Outer Joins

Two flat collection

coll1
```
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
```

coll2
```
[{B:b1, C:c1}, {B:b1, C:c2}, {B:b3, C:c3}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 AS x LEFT 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},
 {A:a1, B:b2}]
```

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

Splitting

country

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

river

```
[{"name": "Donau",    "-country": "SRB A D H HR SK BG AL MD UA"},
 {"name": "Colorado", "-country": "MEX USA"},
  ... ]
```

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

String

Separator

A collection

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

country

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

Splitting

river

```
[{"name": "Donau",    "-country": "SRB A D H HR SK BG AL MD UA"},
 {"name": "Colorado", "-country": "MEX USA"},
  ... ]
```

```
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",    "gdp_total": 4100, "z": "AL"},
  ... ]
```

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

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	b1
3	b2
3	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}]}]
```

SQL++

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

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	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}]}]
```

SQL++

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

Answer:

A	B
a1	b1
a1	b2
a1	b6

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	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}]}]
```

SQL++

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

Answer:

A	B
a1	b1
a1	b2
a1	b6

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	b3

SQL

```
SELECT x.A, y.B
FROM coll AS x, F AS y
WHERE x.id = y.parent AND x.A = “a1”
```

Flattening SQL++ Queries

A nested collection

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

SQL++

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

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	b3

SQL

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}]}]
```

SQL++

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

Answer:

A	B
a1	b1

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	b3

SQL

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}]}]
```

SQL++

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

Answer:

A	B
a1	b1

Relational representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	b1
3	b2
3	b3

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
```

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 for *data exchange*

"General" guidelines:

- For quick, ad-hoc data analysis, query it directly in the native format (Json/SQL++)
- 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