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

Announcmenets

- HW3 (Azure) due tonight (Friday)
- WQ4 (Relational algebra) due Tuesday
- WQ5 (Datalog) due next Friday
- HW4 (Datalog/Logicblox/Cloud9) is posted

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

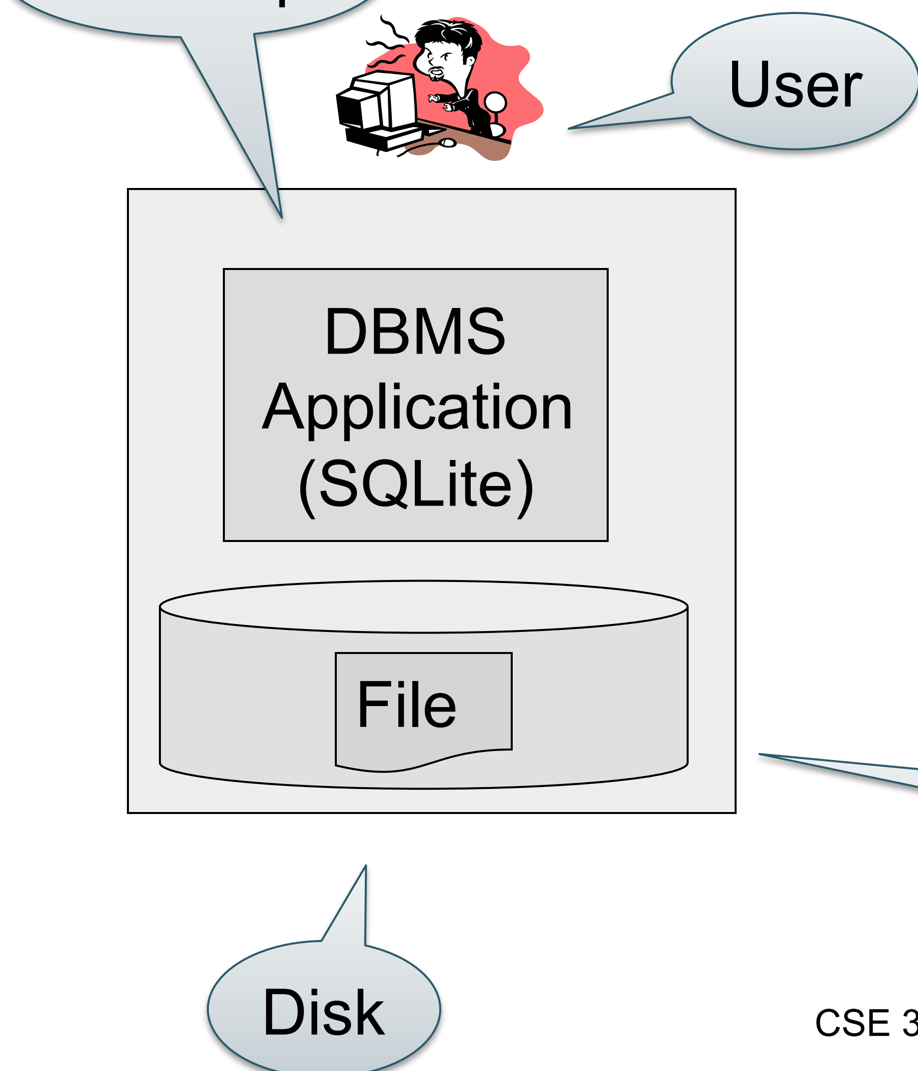
NoSQL Motivation

- Originally motivated by Web 2.0 applications
  - E.g. Facebook, Amazon, Instagram, etc
  - Web startups need to scaleup from 10 to 100000 users very quickly
- Needed: very large scale OLTP workloads
- Give up on consistency
- Give up OLAP

What is the Problem?

- Single server DBMS are too small for Web data
- Solution: scale out to multiple servers
- This is hard for the *entire* functionality of DMBS
- NoSQL: reduce functionality for easier scale up
  - Simpler data model
  - Very restricted updates

RDBMS Review: Serverless

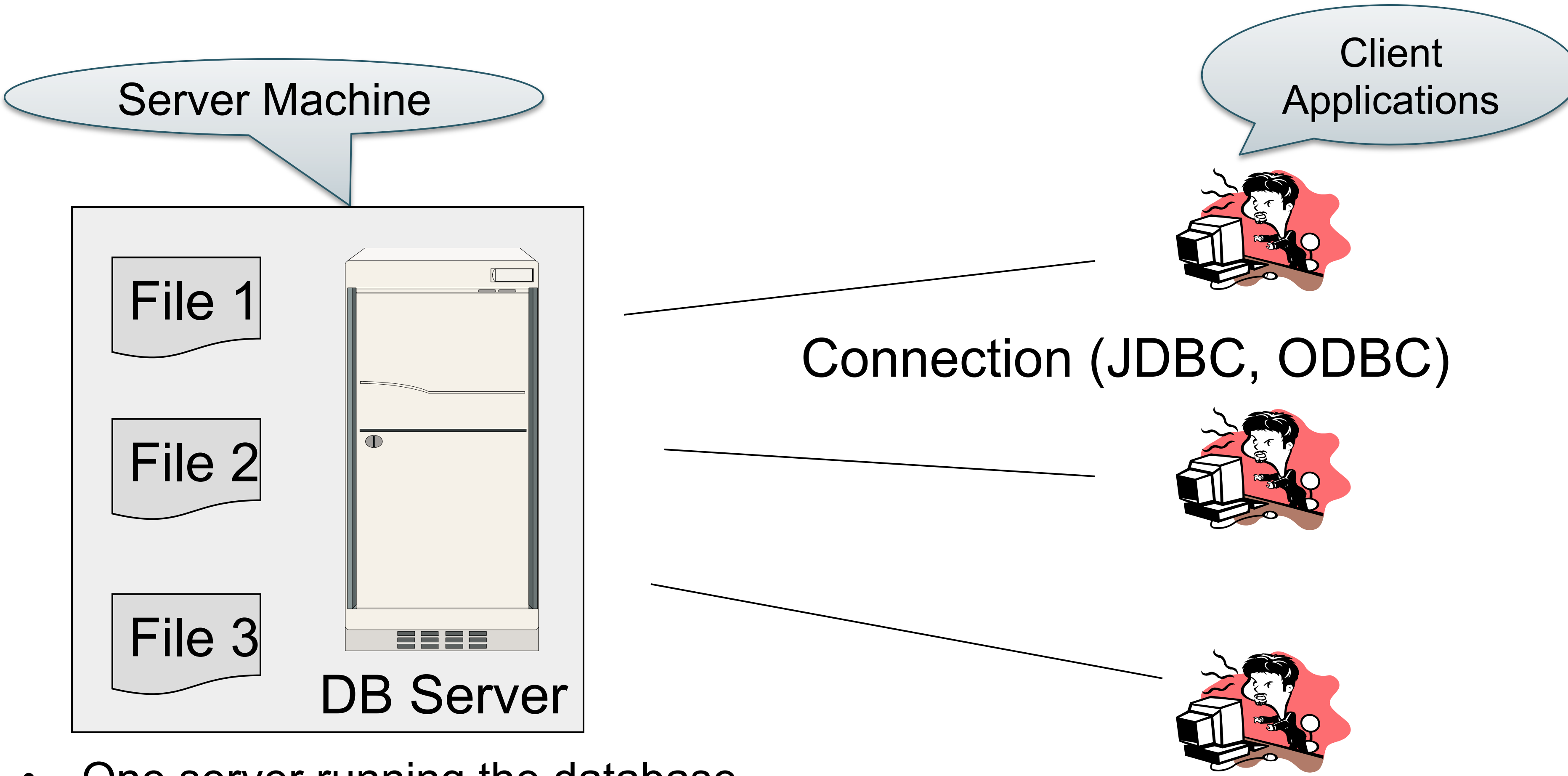


SQLite:

- One data file
- One user
- One DBMS application

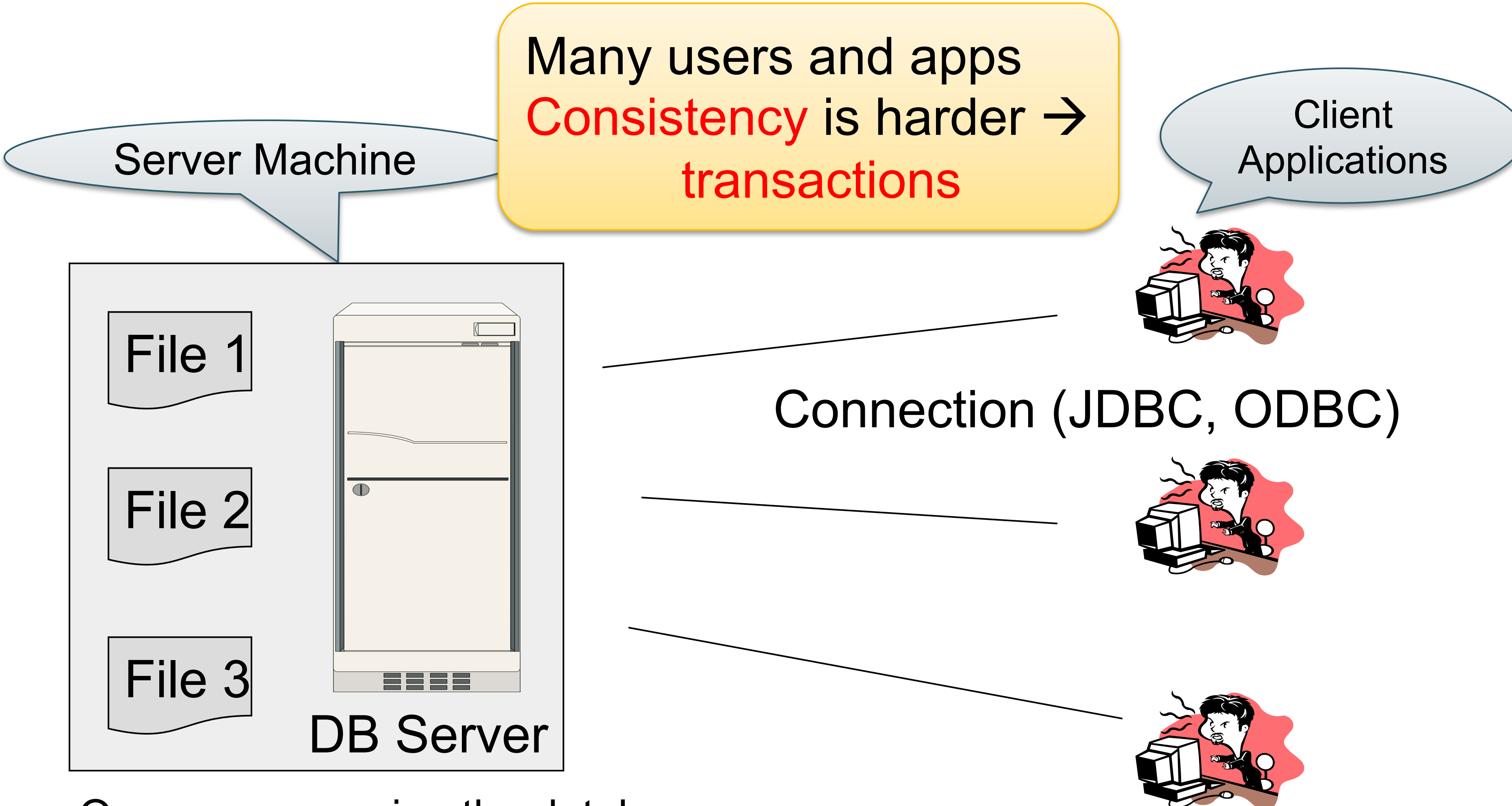
- Consistency is easy
- But only a limited number of scenarios work with such model

RDBMS Review: Client-Server



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

RDBMS Review: 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

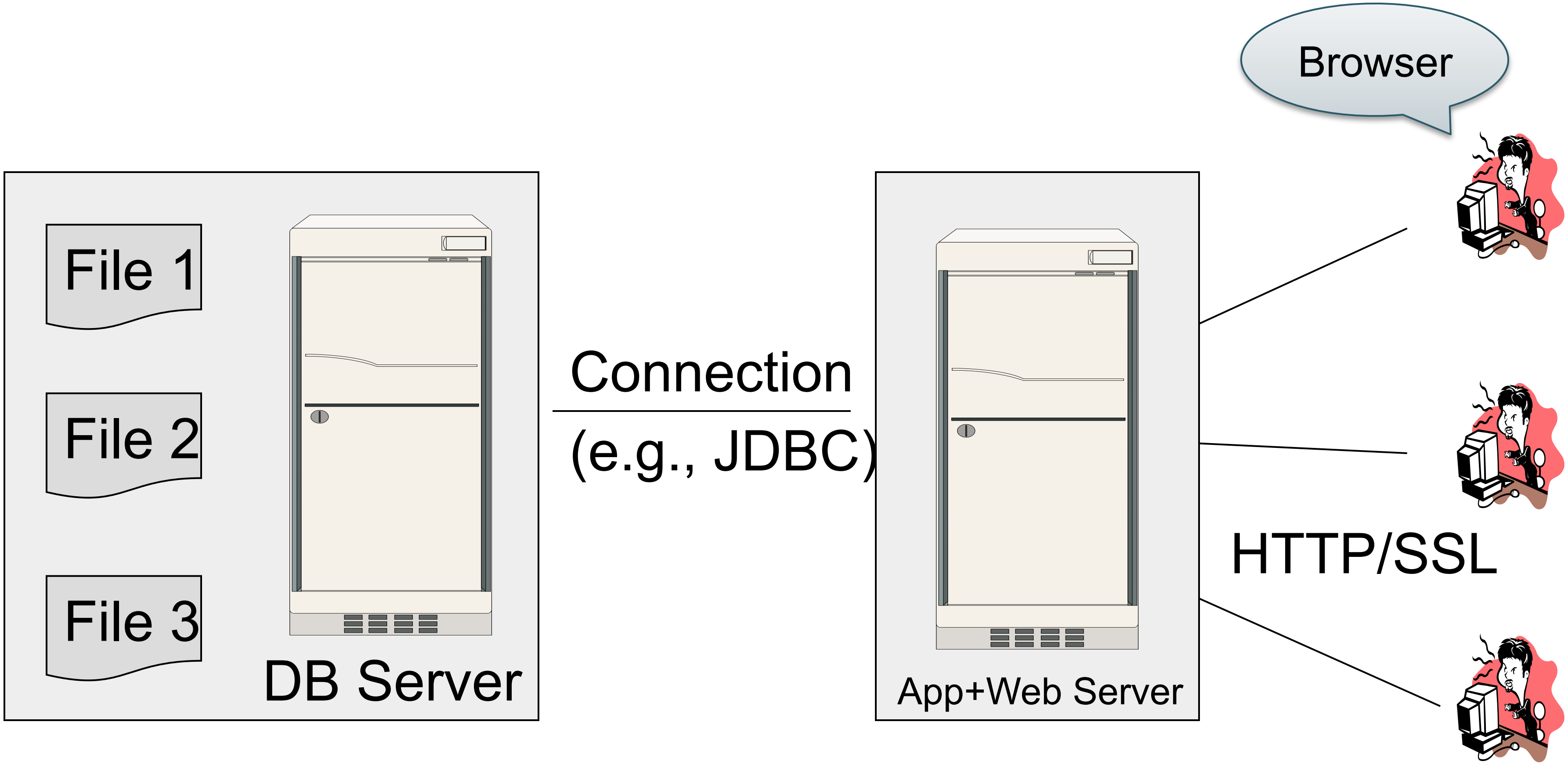
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft's Management Studio (for SQL Server), or
  - psql (for postgres)
  - Some Java program (HW8) or some C++ program

Client-Server

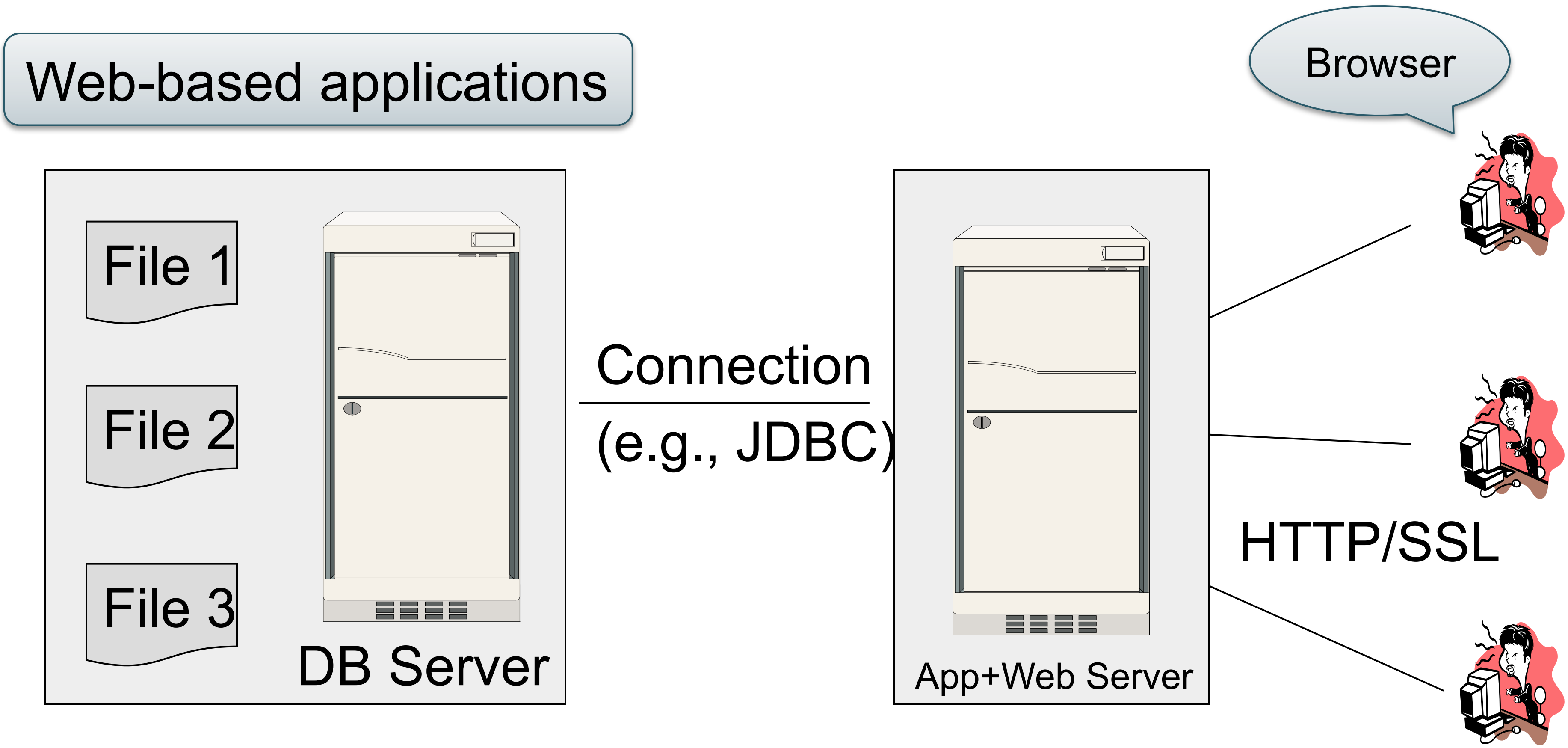
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft's Management Studio (for SQL Server), or
  - psql (for postgres)
  - Some Java program (HW8) or some C++ program
- Clients "talk" to server using JDBC/ODBC protocol

Web Apps: 3 Tier

Web Apps: 3 Tier

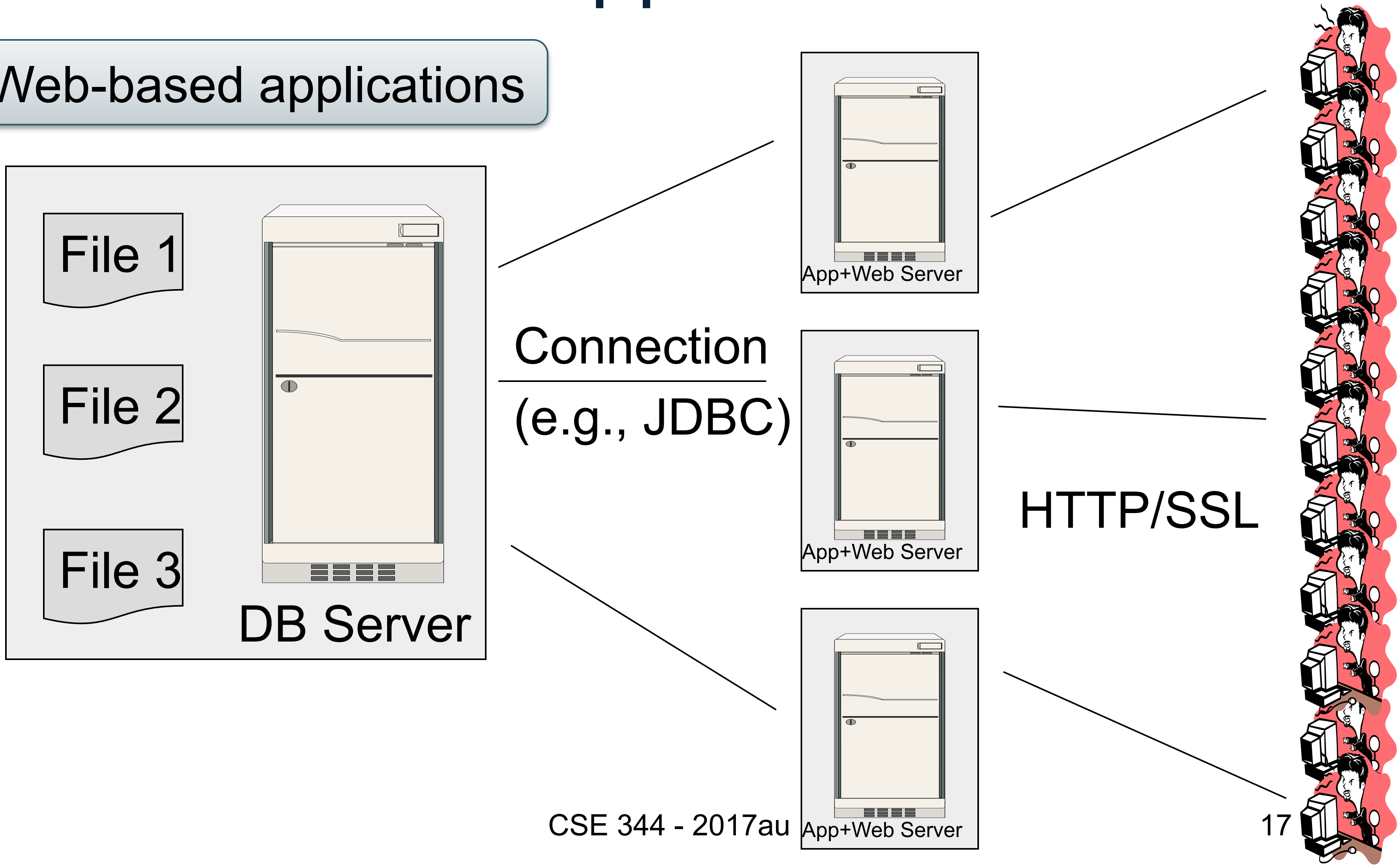


Web Apps: 3 Tier



Web Apps: 3 Tier

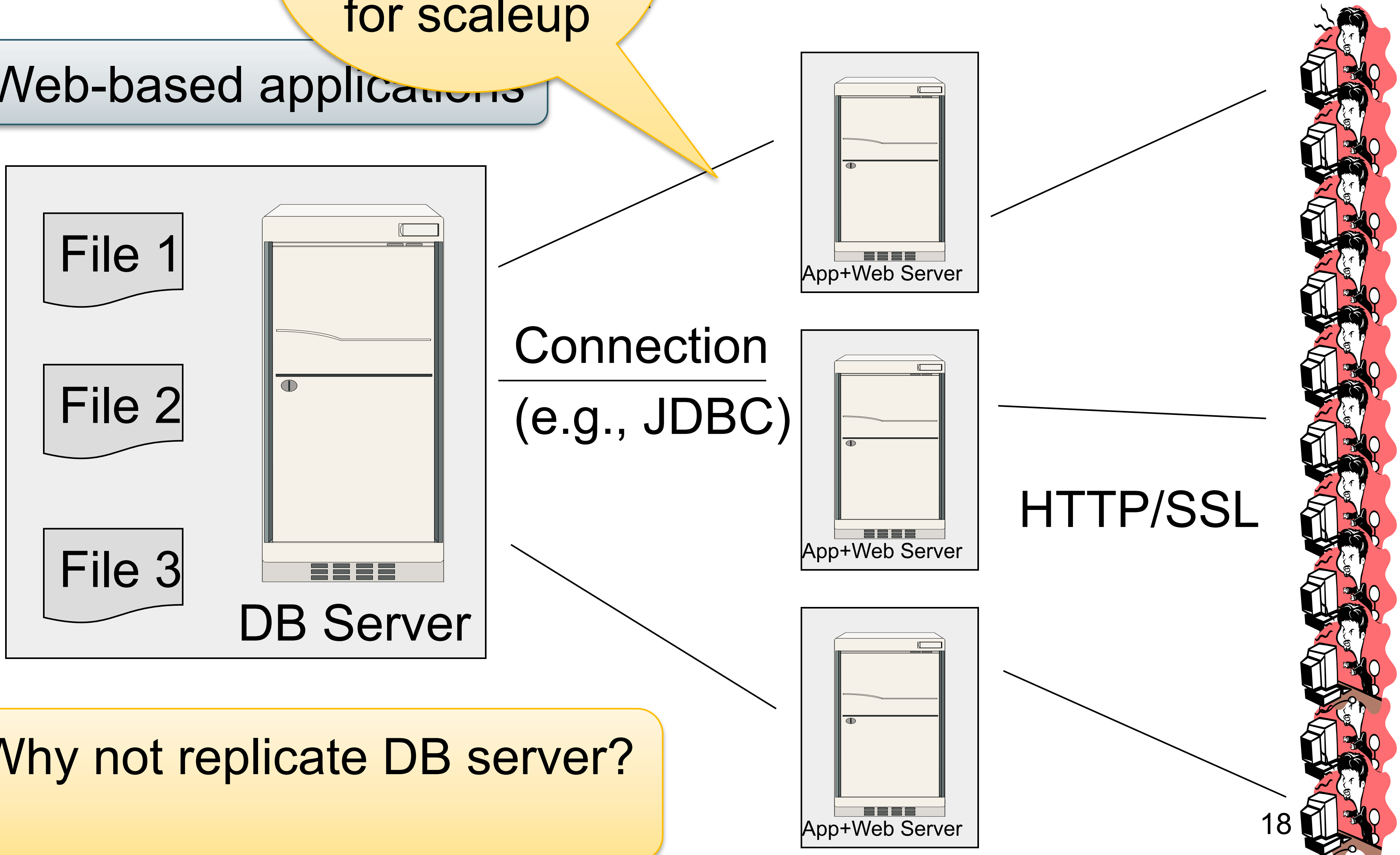
Web-based applications



ps: 3 Tier

Replicate App server for scaleup

Web-based applications

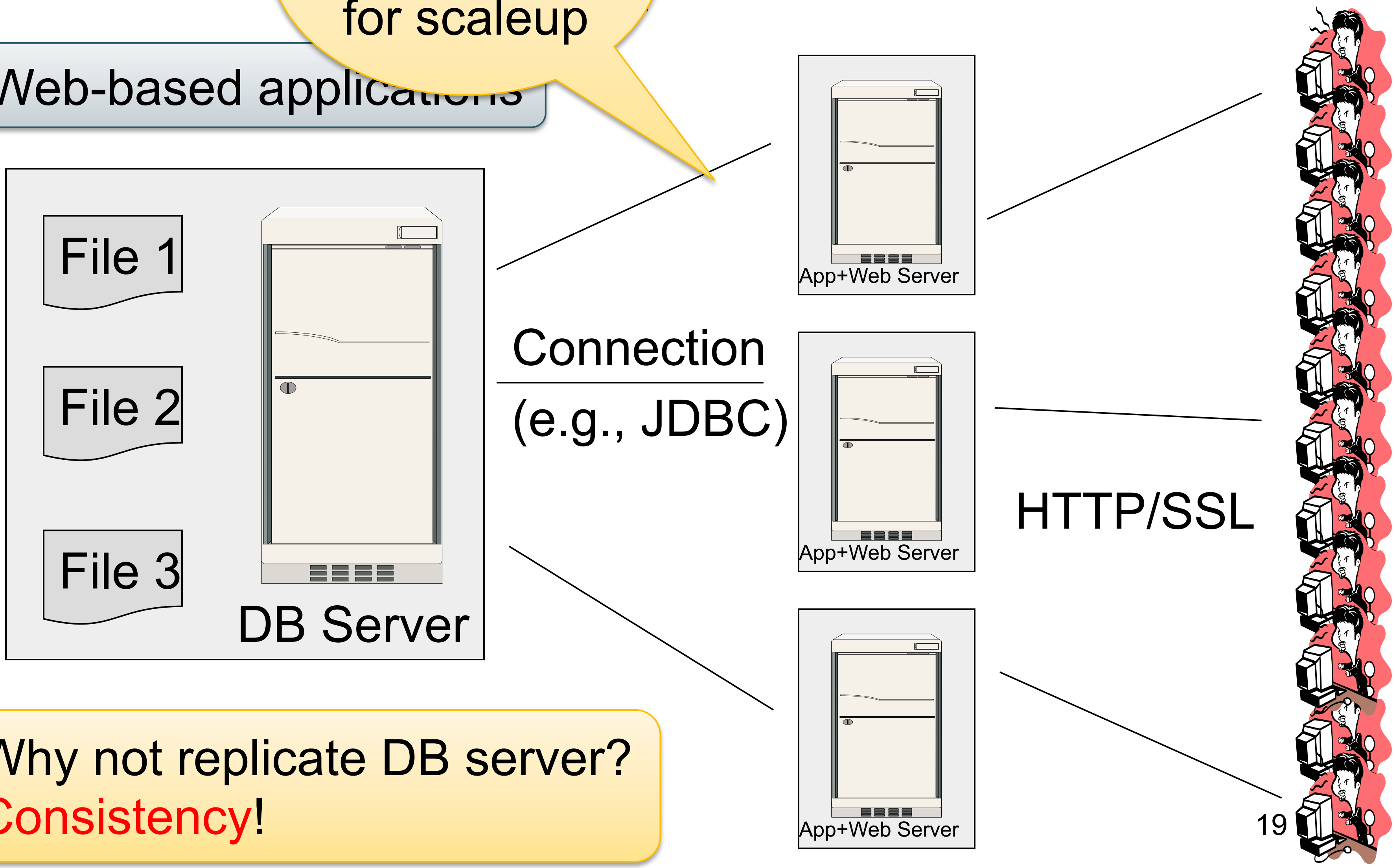


Why not replicate DB server?

ps: 3 Tier

Replicate App server for scaleup

Web-based applications



Why not replicate DB server? Consistency!

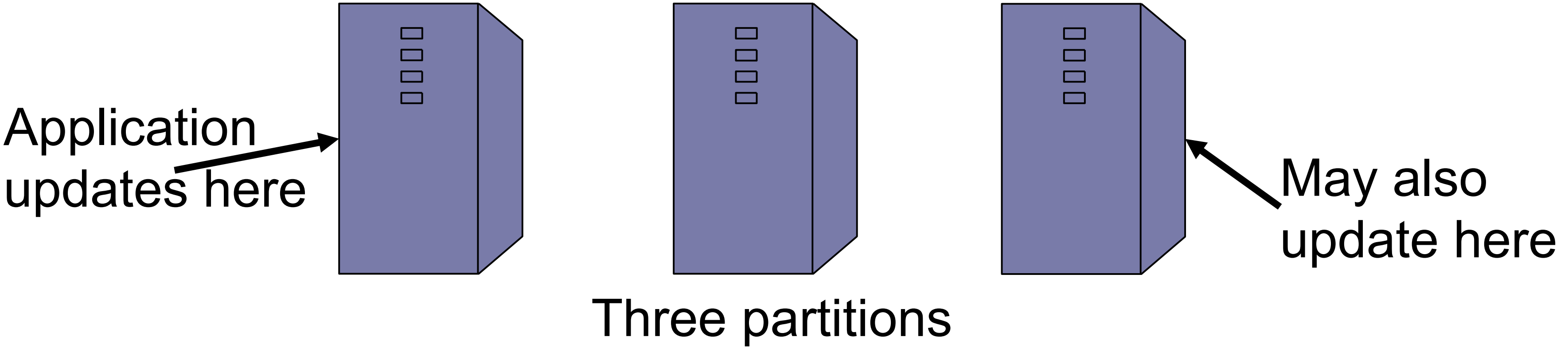
Replicating the Database

- Two basic approaches:
  - Scale up through partitioning
  - Scale up through replication

- Consistency is much harder to enforce

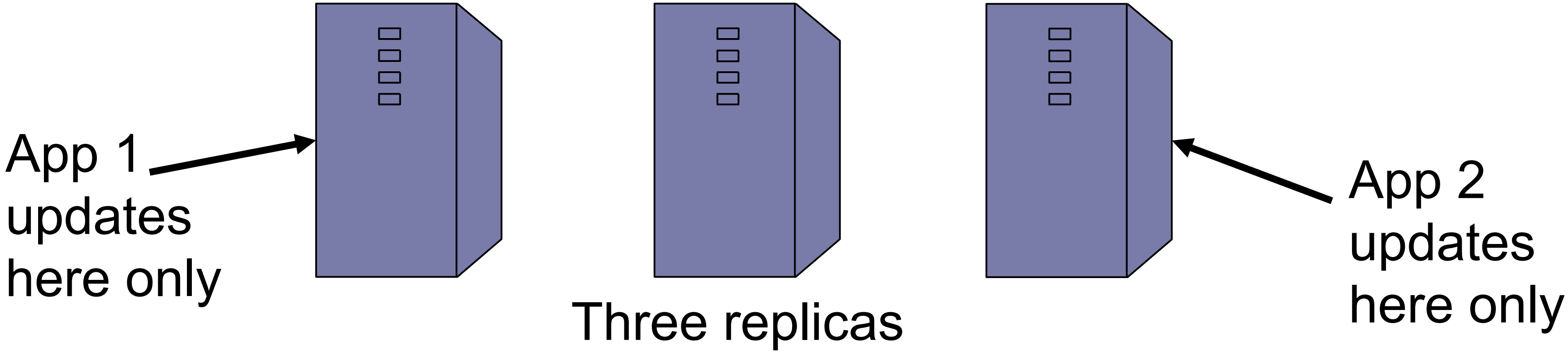
Scale Through Partitioning

- 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
- Given `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, and store every (key,value) pair on server h(key)
  - In class: discuss get(key), and put(key,value)
- 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: allow DBMS to understand the *value*
  - Represent *value* as a JSON (or XML...) document
  - [all flights on that date] = a JSON file
  - May search for all flights on a given date

Document Stores Features

- **Data model**: (key,document) pairs
  - Key = string/integer, unique for the entire data
  - Document = JSon, or XML
- **Operations**
  - Get/put document by key
  - Query language over JSon
- **Distribution / Partitioning**
  - Entire documents, as for key/value pairs

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
- Data model is rows and columns
- Scalability by splitting rows and columns over nodes
  - Rows partitioned through sharding on primary key
  - Columns of a table are distributed over multiple nodes by using "column groups"
- HBase is an open source implementation of BigTable

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 date
- Review session: Friday, 5:30pm, SMI 205

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 Terminology

- Data is represented in name/value pairs.
- Curly braces hold objects
  - Each object is a list of name/value pairs separated by , (comma)
  - Each pair is a name is followed by ':'(colon) followed by the value
- Square brackets hold arrays and values are separated by ,(comma).

JSon Data Structures

- Collections of name-value pairs:
  - {"name1": value1, "name2": value2, …}
  - The "name" is also called a "key"
- Ordered lists of values:
  - [obj1, obj2, obj3, ...]

Avoid Using Duplicate Keys

The standard allows them, but many implementations don't

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

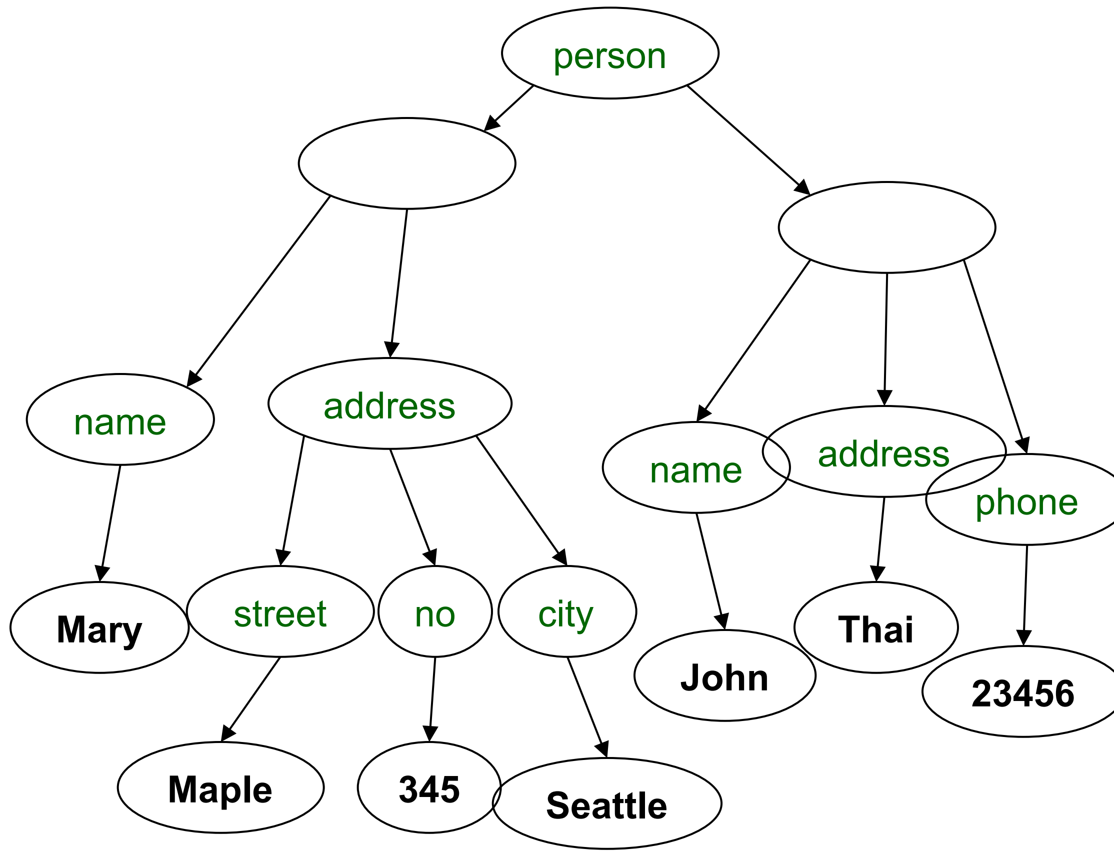
```
{"id":"07",
   "title": "Databases",
   "author": ["Garcia-Molina",
              "Ullman",
              "Widom"]
}
```

JSon Datatypes

- Number
- String = double-quoted
- Boolean = true or false
- nullempty

JSon Semantics: a Tree !

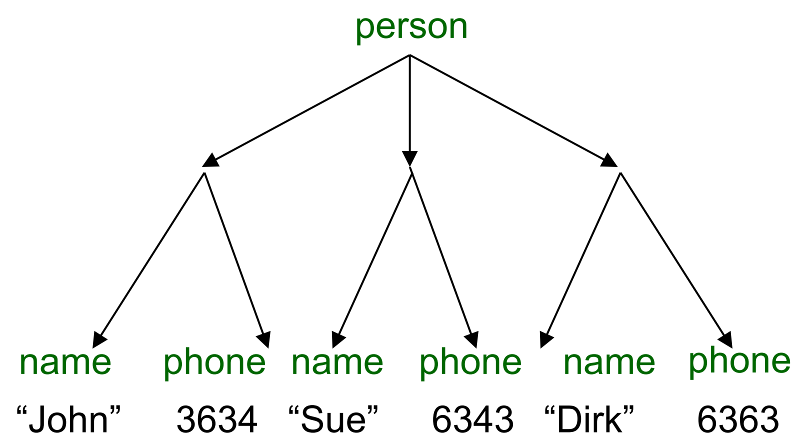
```
{"person":
   [ {"name": "Mary",
      "address":
         {"street":"Maple",
           "no":345,
           "city": "Seattle"}},
    {"name": "John",
      "address": "Thailand",
      "phone":2345678}}
    ]
}
```



JSon Data

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

Mapping Relational Data to JSon



Person

name	phone
John	3634
Sue	6343
Dirk	6363

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

Mapping Relational Data to JSon

May inline foreign keys

Person

name	phone
John	3634
Sue	6343

Orders

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

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

JSon=Semi-structured Data (1/3)

- Missing attributes:

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

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	-

JSon=Semi-structured Data (2/3)

- Repeated attributes

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

Two phones !

- Impossible in one table:

name	phone	
Mary	2345	3456

???

JSon=Semi-structured Data (3/3)

- Attributes with different types in different objects

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

Structured name !

- Nested collections
- Heterogeneous collections

Discussion

- *Data exchange formats*
  - 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 end of lecture, 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, may be replaced by AQL (better designed)
  - Asterix: SQL++ (based on SQL)
  - 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~~}

Can’t have repeated fields

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

Examples

Try these queries:

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

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

Can only select from multi-set or array

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

Datatypes

- Boolean, integer, float (various precisions), geometry (point, line, …), date, time, etc
- UUID = universally unique identifier
  Use it as a system-generated unique key

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}] x;
```

```
{ "b": { "int64": 2 } }
{  }
```

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

```
{ "b": { "int64": 2 } }
{  }
```

SQL++ Overview

- DDL: create a
  - Dataverse
  - Type
  - Dataset
  - Index

- DML: select-from-where

Dataverse

A Dataverse is a Database

```
CREATE DATAVERSE lec344
CREATE DATAVERSE lec344 IF NOT EXISTS

DROP DATAVERSE lec344
DROP DATAVERSE lec344 IF EXISTS

USE lec344
```

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 lec344;
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 lec344;
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 lec344;
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 lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   email: string?
}
```

{“Name”: “Alice”}
{“Name”: “Bob”}
…

```
USE lec344;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

Dataset with Auto Generated Key

```
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   myKey: uuid,
   Name : string,
   email: string?
}
```

{“Name”: “Alice”}
{“Name”: “Bob”}
…

Note: no myKey since it will be autogenerated

```
USE lec344;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;
```

Discussion of NFNF

- NFNF = Non First Normal Form
- One or more attributes contain a collection
- One extreme: a single row with a huge, nested collection
- Better: multiple rows, reduced number of nested collections

Example from HW5

mondial.adm is totally semistructured:
{“mondial”: {“country”: [...], “continent”:[...], ..., “desert”:[...]}}

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

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

Country:

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

Indexes

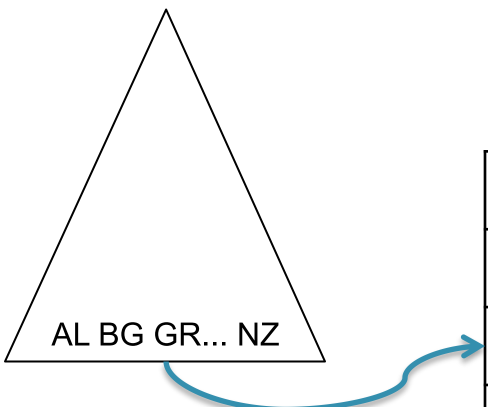
- Can declare an index on an attribute of a top-most collection
- Available:
  - BTREE: good for equality and range queries
    E.g. name="Greece";   20 < age and age < 40
  - RTREE: good for 2-dimensional range queries
    E.g. 20 < x and x < 40 and 10 < y and y < 50
  - KEYWORD: good for substring search

Indexes

Cannot index inside a nested collection

```
USE lec344;
CREATE INDEX countryID
    ON country(`-car_code`)
    TYPE BTREE;
```

```
USE lec344;
CREATE INDEX cityname
    ON country(city.name)
    TYPE BTREE;
```



Country:

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

Introduction to Data Management CSE 344

Lecture 13: SQL++

SQL++ Overview

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

Retrieve Everything

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

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

Answer

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

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

Retrieve countries

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

Answer

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

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

Retrieve countries, one by one

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

Answer

country1
country2
...

Escape characters

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

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

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

Answer

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

Nested Collections

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

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

x.B is a collection

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

Nested Collections

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

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

x.B is a collection

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

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

Heterogeneous Collections

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

Runtime error

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

The problem:

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

city is an array

city is an object

Heterogeneous Collections

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

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE  y.name='Greece' and is_array(z.city);
```

The problem:

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

Just the arrays

Heterogeneous Collections

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

Note: get name directly from z

```
SELECT z.name as province_name, z.city.name as city_name
FROM world x, x.mondial.country y, y.province z
WHERE  y.name='Greece' and not is_array(z.city);
```

Just the objects

The problem:

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

Heterogeneous Collections

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

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
          (CASE  WHEN is_array(z.city) THEN z.city
           ELSE  [z.city] END) u
WHERE  y.name='Greece';
```

Get both!

The problem:

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

Heterogeneous Collections

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

```
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:

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

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 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}]}]
```

Nested Relational Algebra

$Unnest_F(coll)$ =

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

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}]}]
```

Nested Relational Algebra

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}]}]
```

SQL++

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

Refers to relations defined on the left

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}]}]
```

Nested Relational Algebra

$Unnest_F$(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}]}]
```

SQL++

```
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}]}]

Nested Relational Algebra

$Unnest_F$(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]}

SQL++

```
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}]}]

Nested Relational Algebra

$Unnest_F$(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]}

SQL++

```
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 (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)
```

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

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
```

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

Function	NULL	MISSING	Empty Collection
COLL_COUNT	counted	counted	0
COLL_SUM	returns NULL	returns NULL	returns NULL
COLL_MAX	returns NULL	returns NULL	returns NULL
COLL_MIN	returns NULL	returns NULL	returns NULL
COLL_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

Group-by / Aggregate

A flat collection

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

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

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

Are these equivalent?

Group-by / Aggregate

A flat collection

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

```
SELECT DISTINCT x.A, COLL_COUNT(g) as cnt
FROM coll x
LET
```

Lesson: Read the *$@# manual!!

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

Are these equivalent?

Join

Two flat collection

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

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

Multi-Value Join

- 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

Multi-Value Join

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

Multi-Value Join

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

String

Separator

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

Multi-Value Join

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

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

String

Separator

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

Behind the Scenes

Query Processing on NFNF data:

- Option 1: give up on query plans, 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

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}]}]
```

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:

id	A
1	a1
2	a2
3	a1

F

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

G

parent	C
1	c1
3	c2
3	c3

Flattening SQL++ Queries

A nested collection

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

SQL++

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

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

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

G

parent	C
1	c1
3	c2
3	c3

SQL

Flattening SQL++ Queries

A nested collection

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

SQL++

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

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

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

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = ‘a1’
```

Flattening SQL++ Queries

A nested collection

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

SQL++

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

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

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

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

G

parent	C
1	c1
3	c2
3	c3

SQL

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

Flattening SQL++ Queries

A nested collection

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

SQL++

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

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

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

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

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = ‘a1’
```

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

Semistructured Data Model

- Several file formats: Json, protobuf, XML
- The data model is a tree
- They differ in how they handle structure:
  - Open or closed
  - Ordered or unordered

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