CSE 344

JULY 9TH

NOSQL

ADMINISTRATIVE MINUTIAE

- **HW3 due Wednesday**
  - tests released
  - actual_time should have 0s not NULLs
    - upload new data file
    - or use UPDATE to change 0 ~> NULL
- **Extra OOs on Mondays 5-7pm**
  - in CSE 006 (Andrew)
- **Recording lectures**
  - email me for missed class to get access

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

Term has two different meanings

1. non-relational data (more useful)
2. simplified functionality (less useful)

Item 2. originally motivated by Web 2.0 applications

- E.g. eBay, Facebook, Amazon, Instagram, etc
- Web startups need to scale up to 100+m users very quickly

Needed: very large scale OLTP workloads

Give up on large-scale consistency

Give up OLAP

WHAT IS THE PROBLEM?

Single server DBMS are too small for Web data

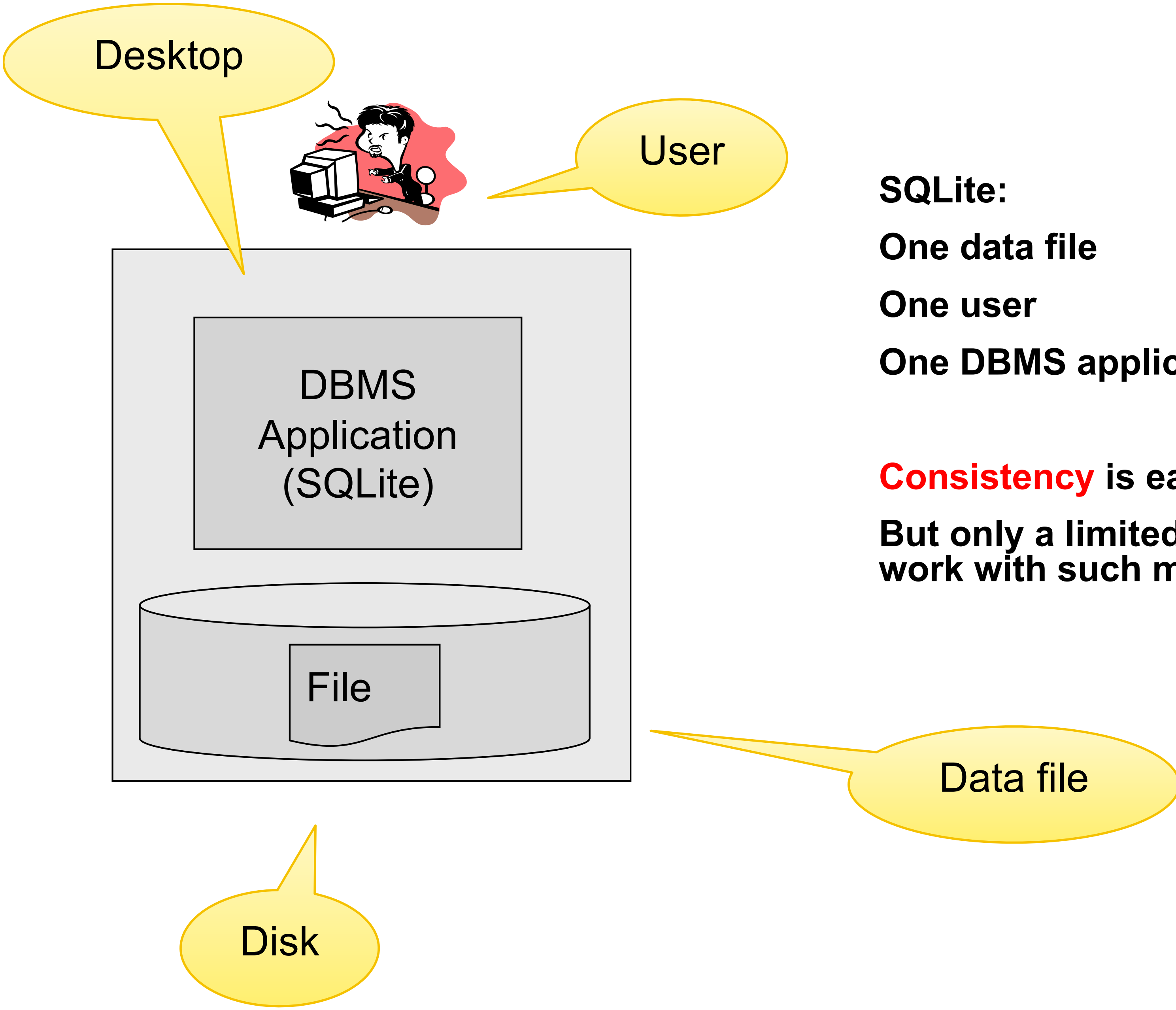
Solution: try scaling out to multiple servers

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

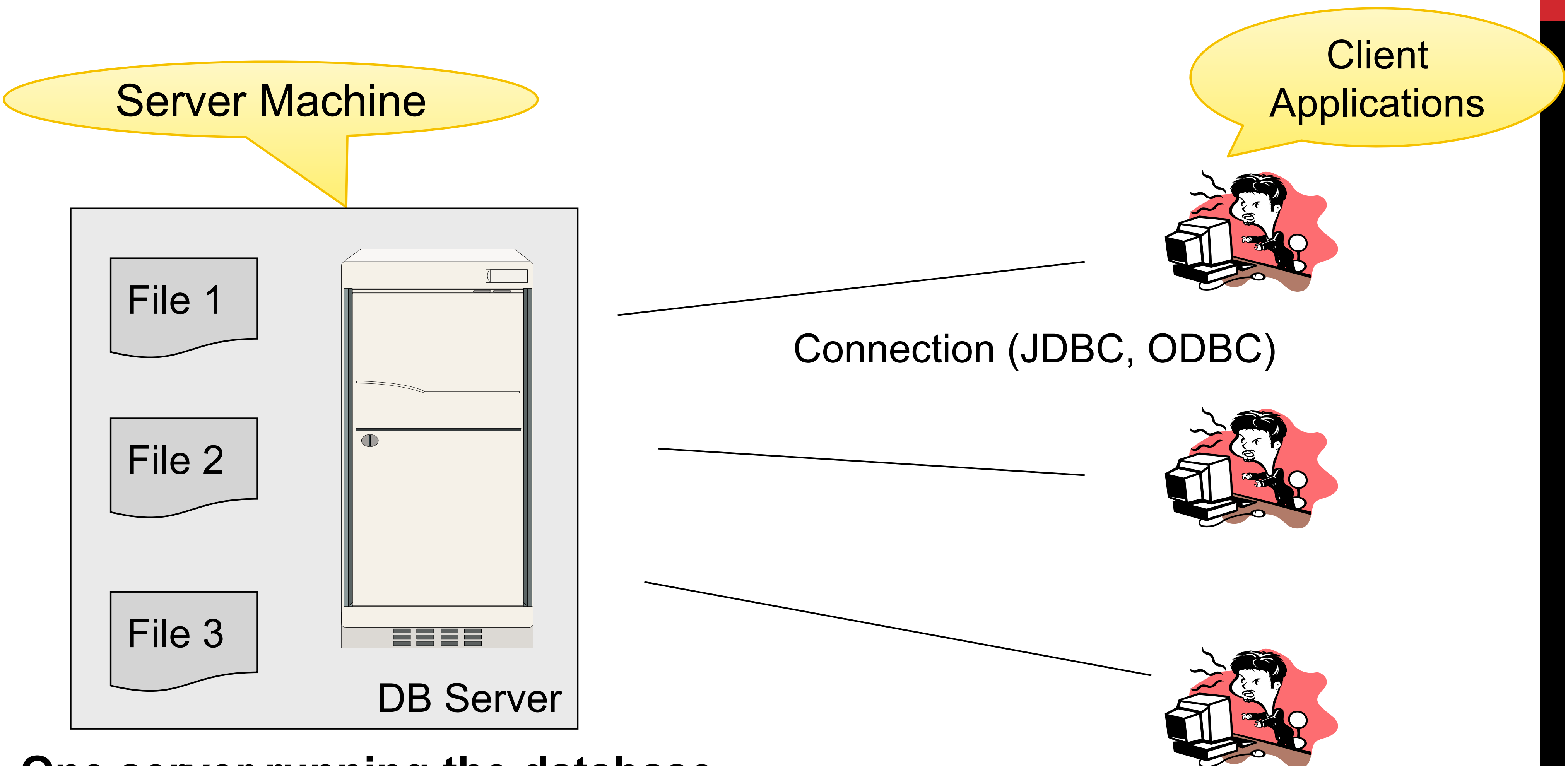
One user

One DBMS application

Consistency is easy

But only a limited number of scenarios work with such model

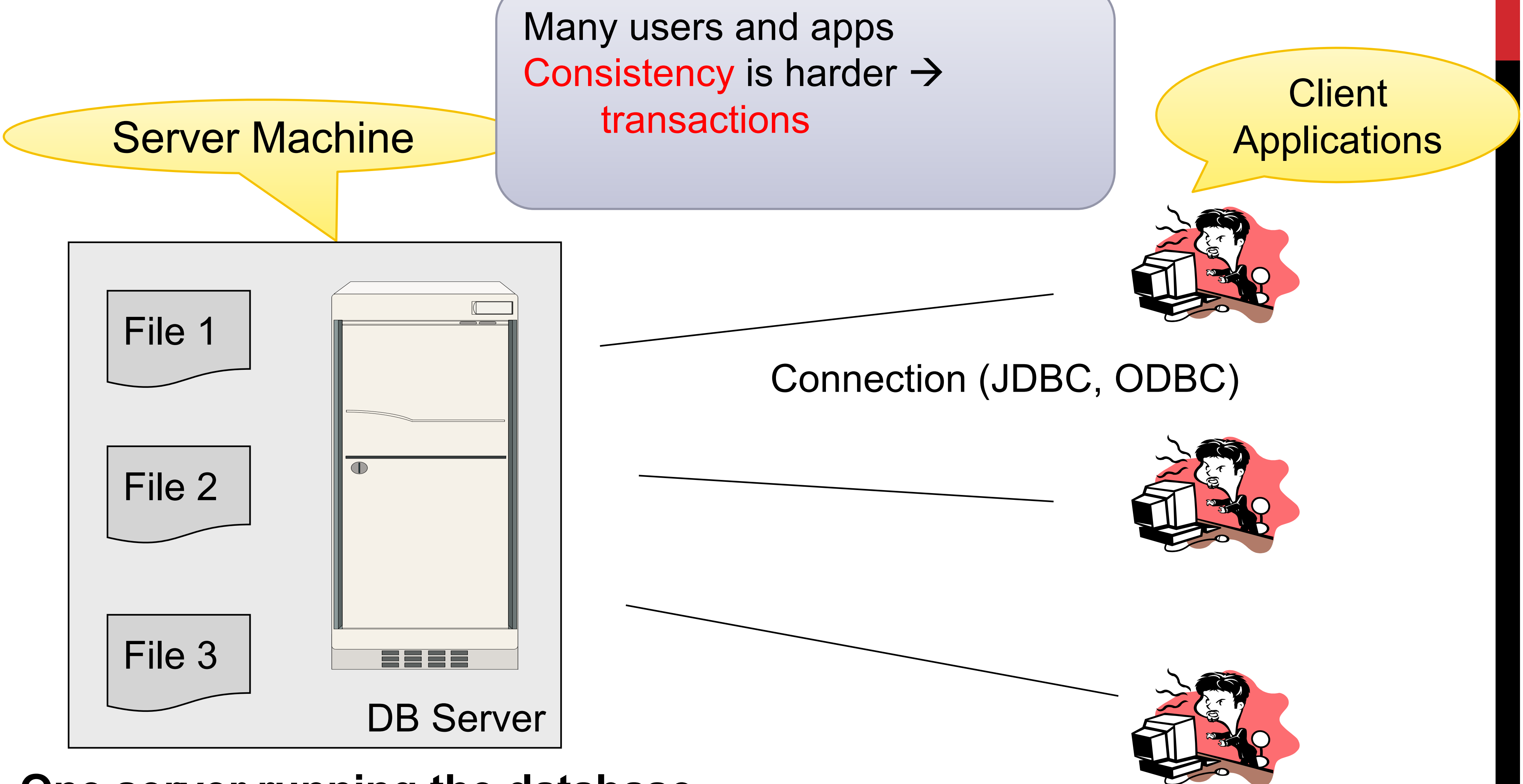
RDBMS REVIEW: CLIENT-SERVER



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):

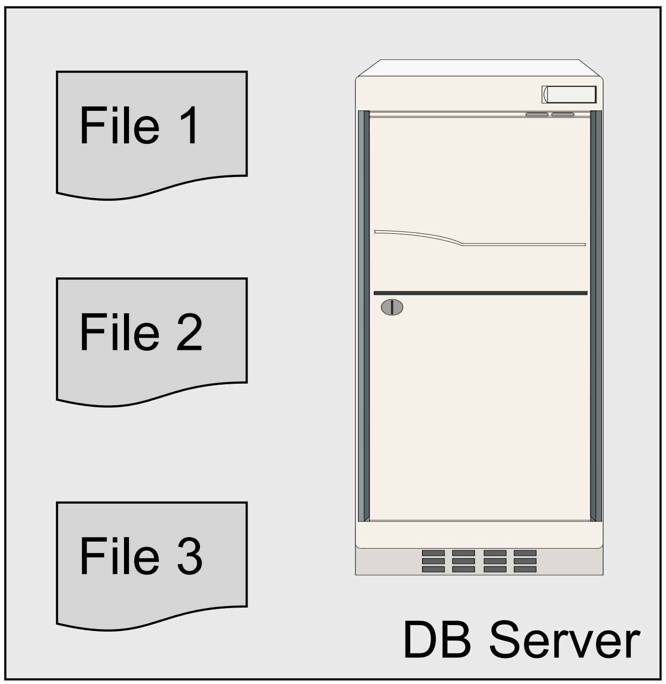
- Your own desktop, or
- Some beefy system, or
- A cloud service (SQL Azure)

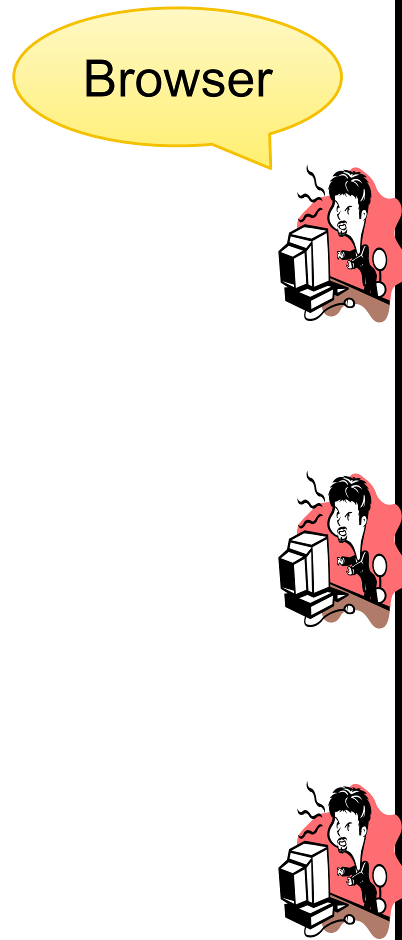
Many *clients* run apps and connect to DBMS

- Microsoft's Management Studio (for SQL Server), or
- psql (for postgres)
- Some Java program (HW8) or some C++ program

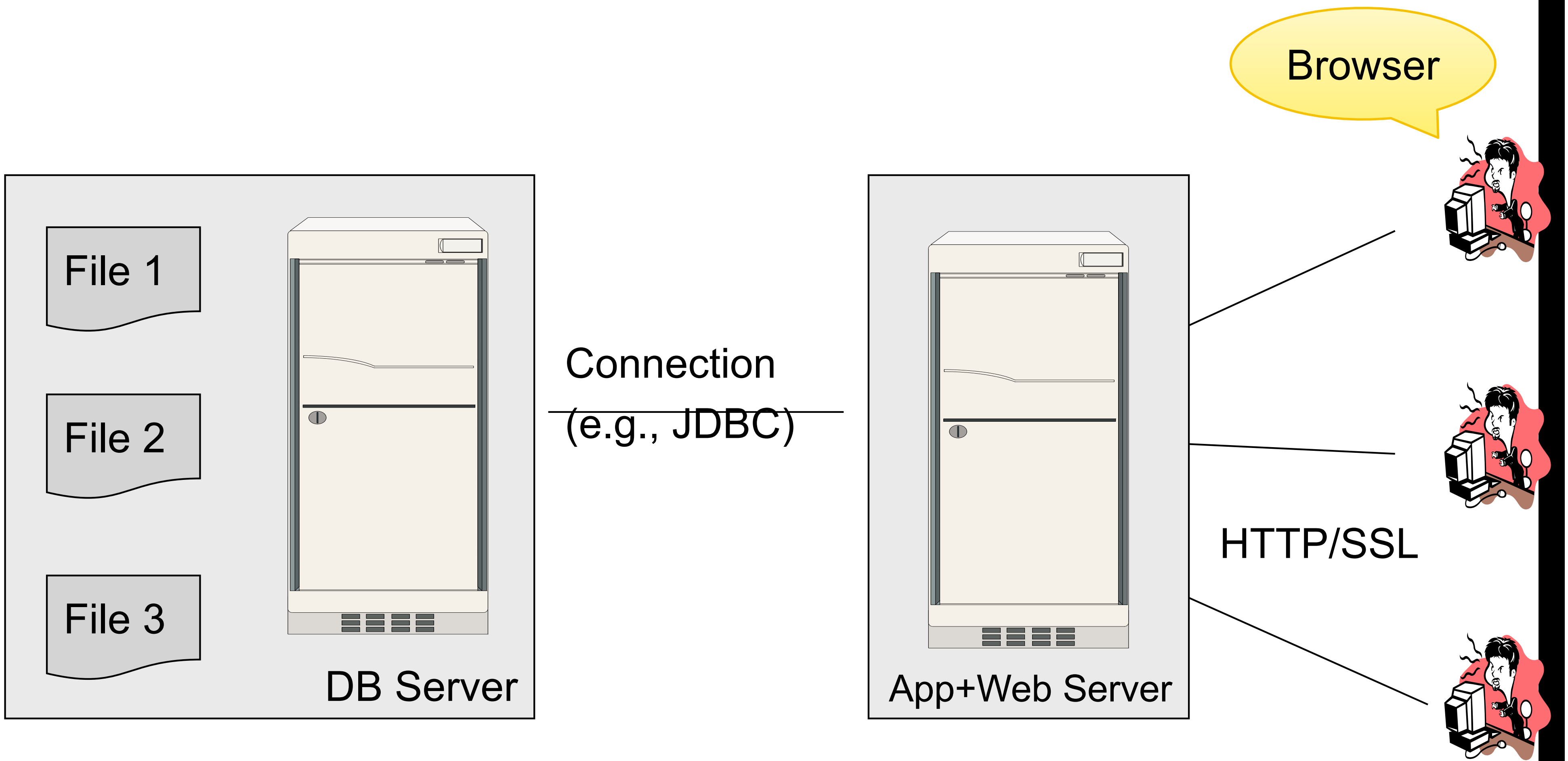
Clients "talk" to server using JDBC/ODBC protocol

WEB APPS: 3 TIER



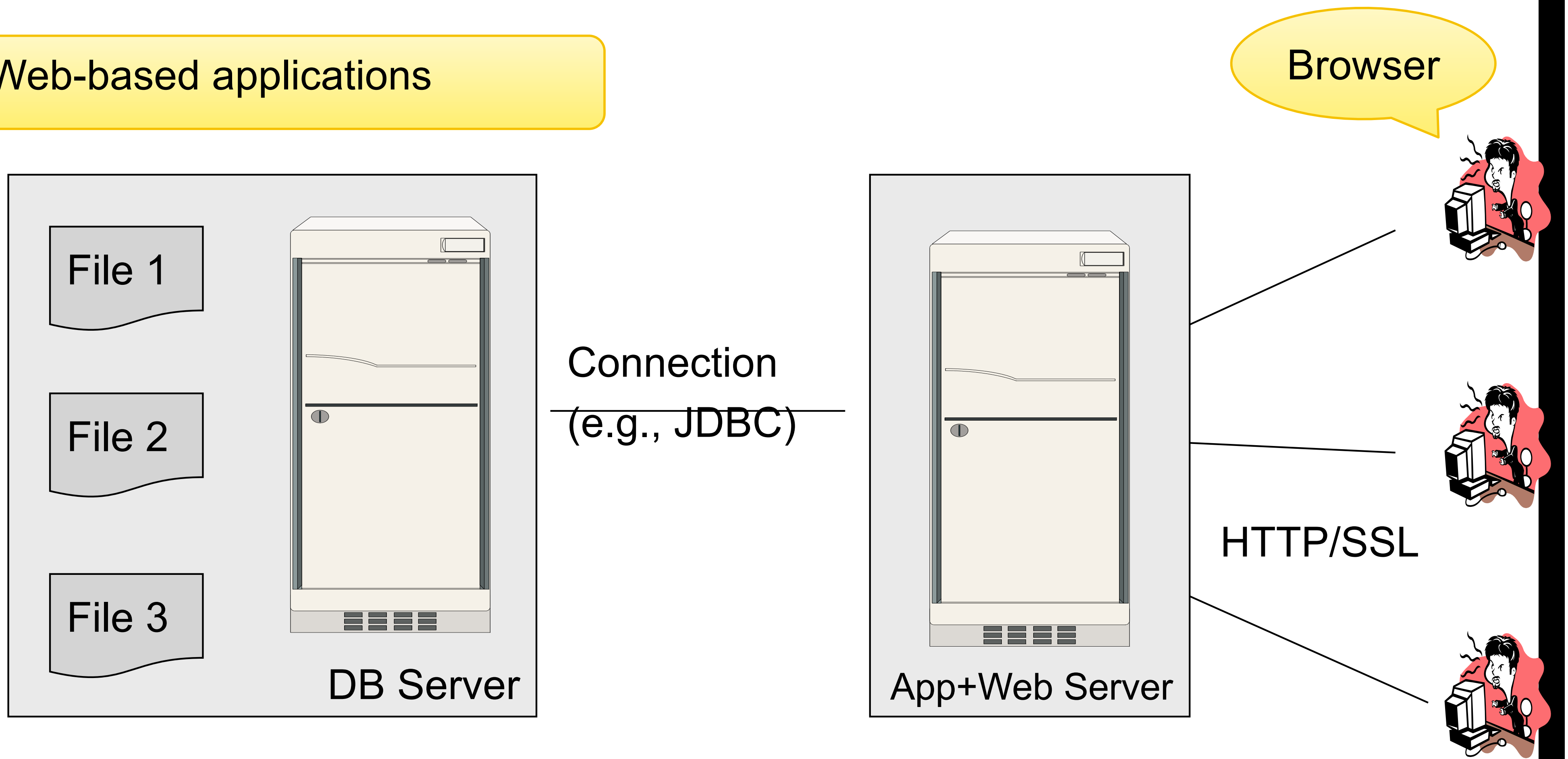


WEB APPS: 3 TIER



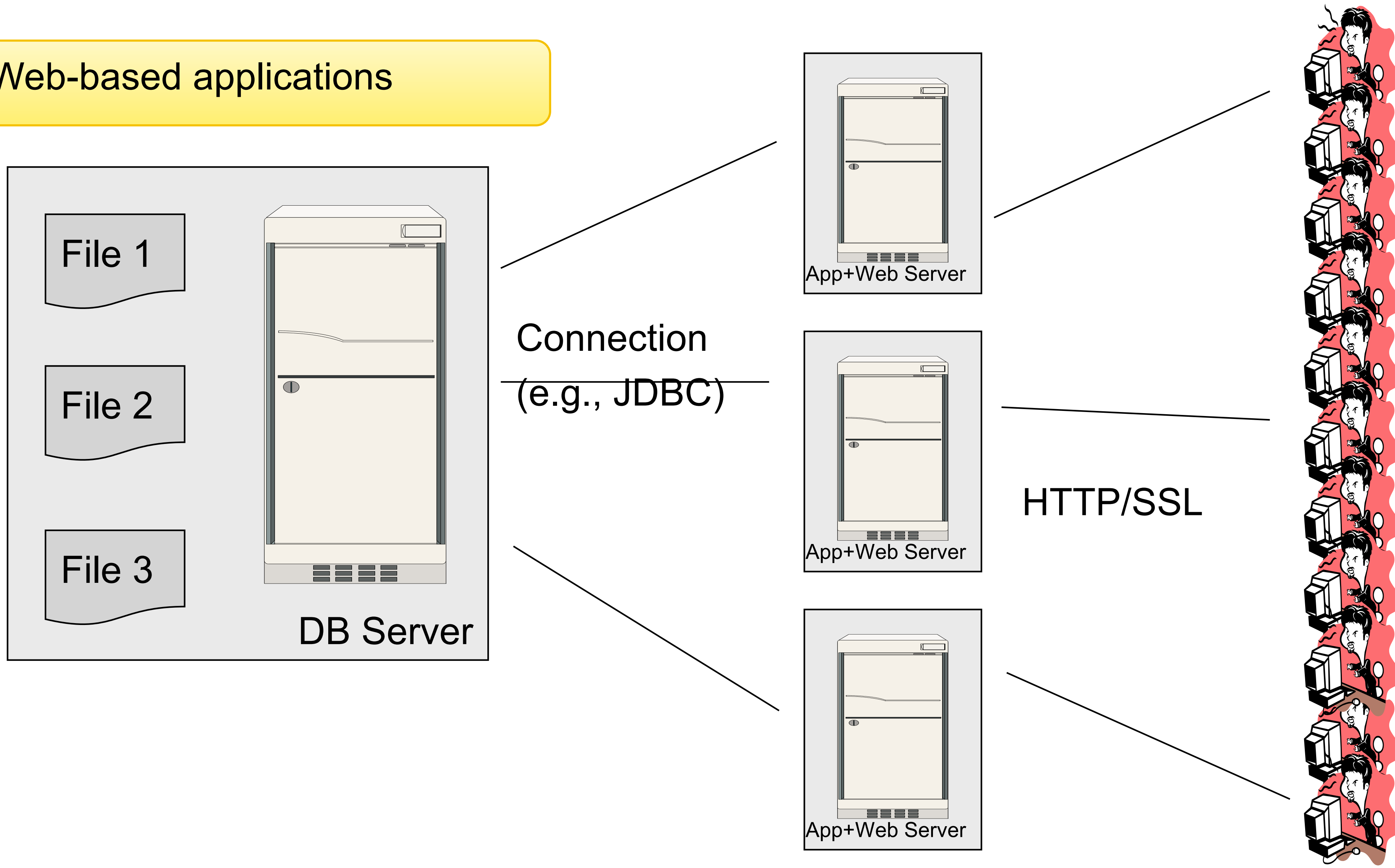
WEB APPS: 3 TIER

Web-based applications



WEB APPS: 3 TIER

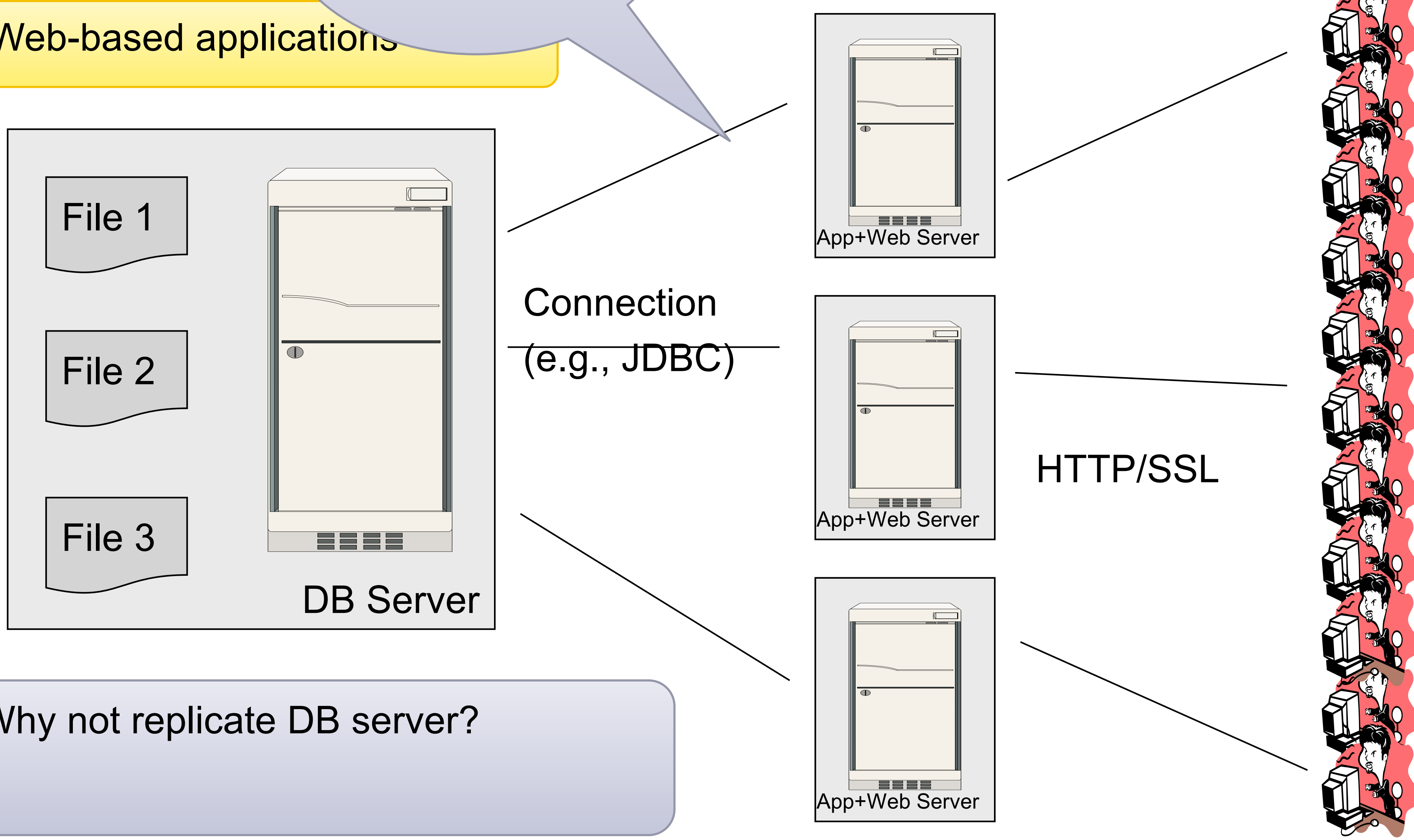
Web-based applications



WEB A… ER

Replicate
App server
for scaleup

Web-based applications

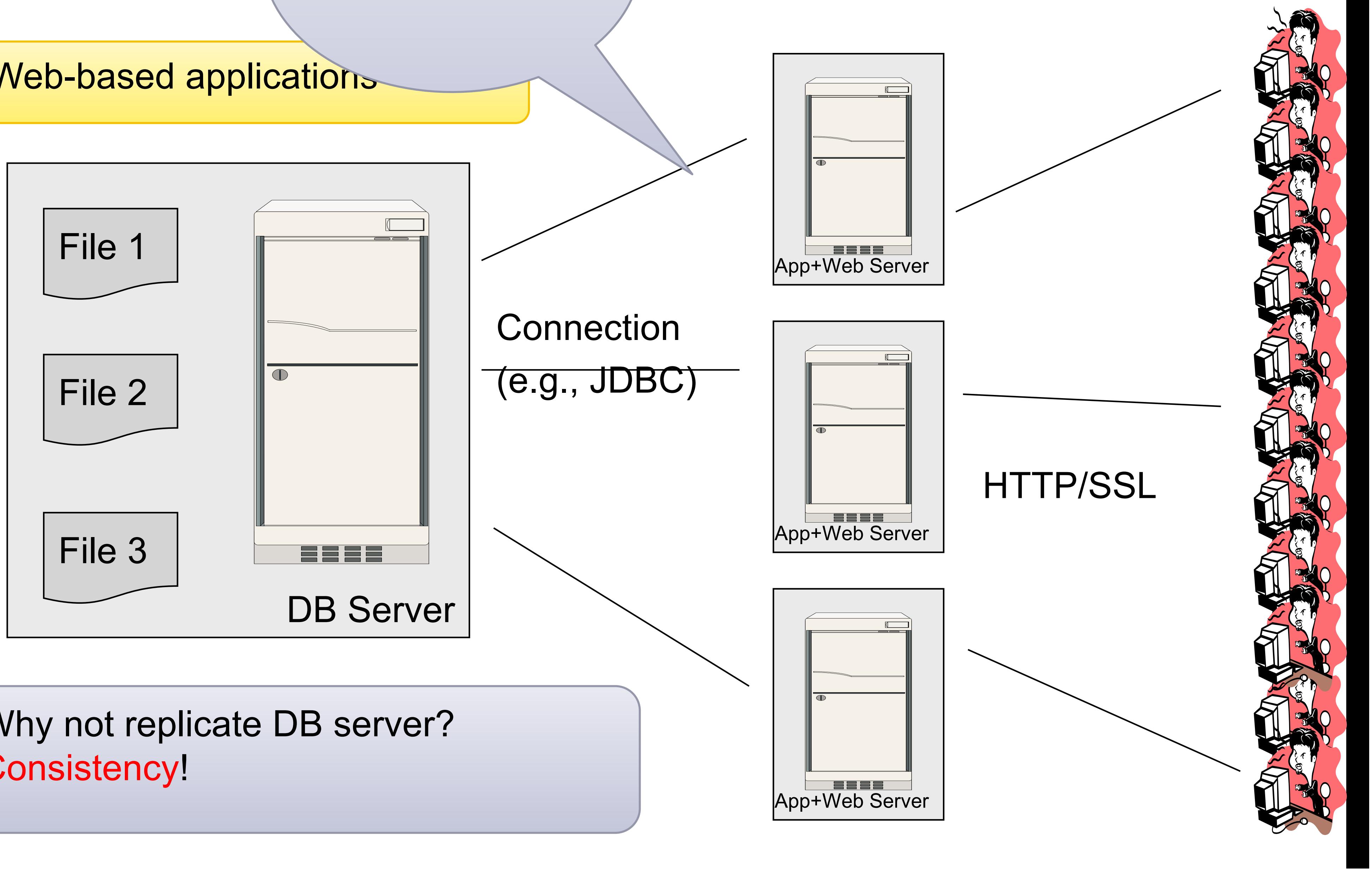


Why not replicate DB server?

WEB A ER

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

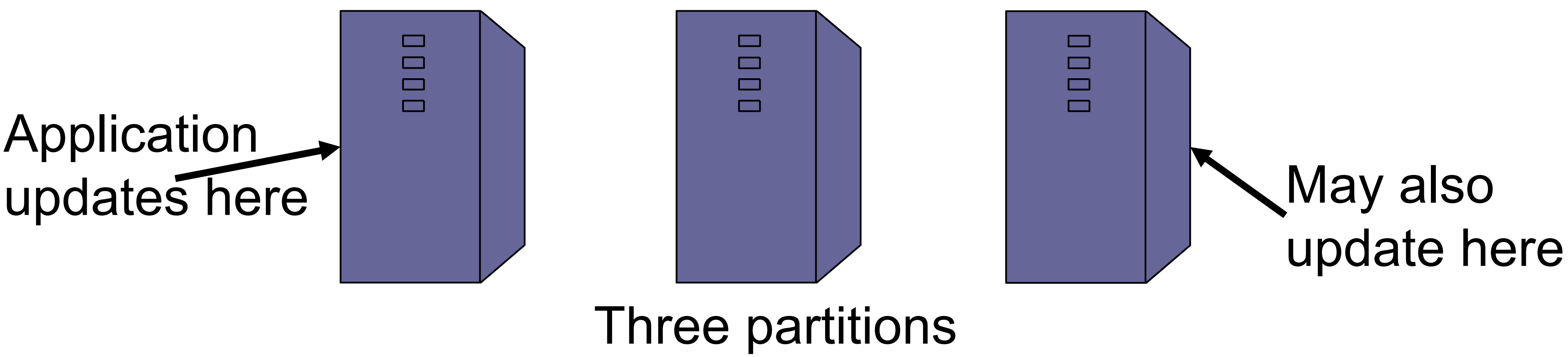
SCALE THROUGH PARTITIONING

Partition the database across many machines in a cluster

- Database now fits in main memory
- Queries spread across these machines

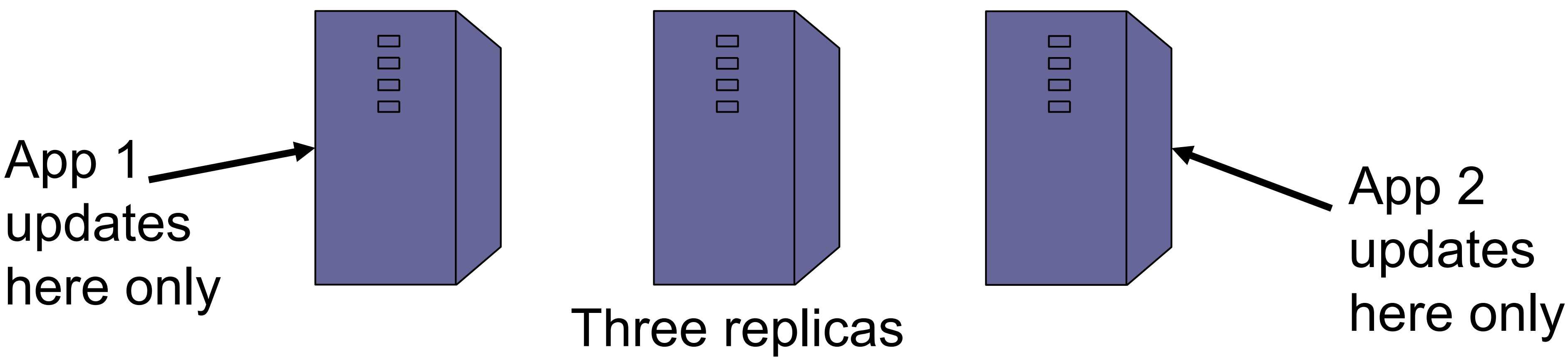
Can increase throughput

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

- Give up much functionality
- Application must now handle joins and consistency
    - (that's a lot!)
- (Future NoSQL systems should fix this.)

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)
    - h(k) returns a number in [0, num_machines-1]
- 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)
    - h(k) returns a number in [0, num_machines-1]
- 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

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

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

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 Algebra

Semistructured data model / JSon

- Flexible, nested structure (trees)
- Does not require predefined schema ("self describing”)
- Text representation: good for exchange, bad for performance
    - not a panacea: more rigid structures are easier for you to query too!
- 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).