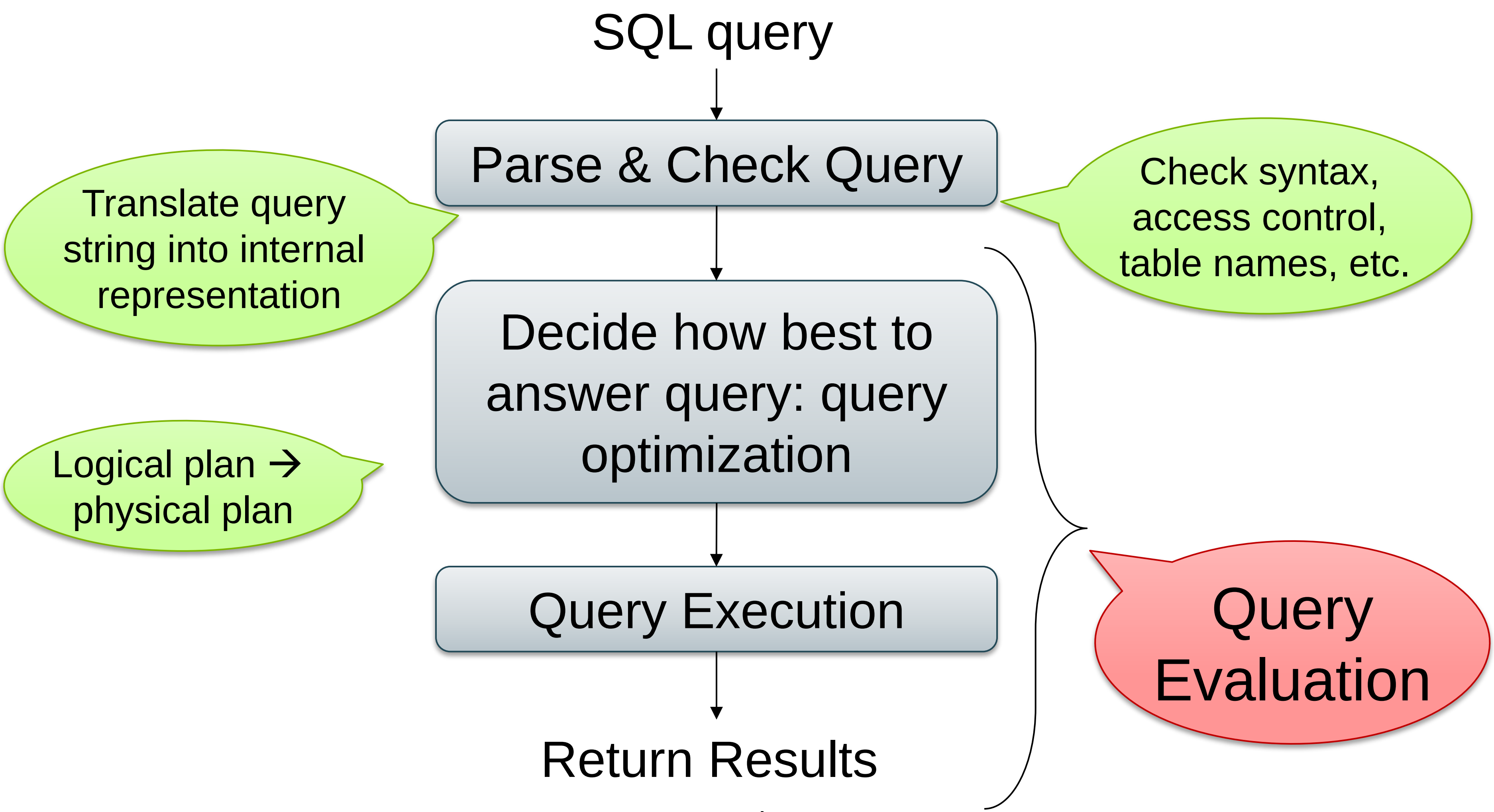
Database Systems
CSE 514

Lectures 06
Size Estimation; NoSQL, JSon

Today

- Database statistics and size estimation
- NoSQL and the semistructured data model

Query Evaluation Steps



Database Statistics

- Collect statistical summaries of stored data
- Estimate size (=cardinality), bottom-up
- Estimate cost by using the estimated size

Database Statistics

- Number of tuples T(R) = cardinality
- Number of distinct values of attribute a V(R,a)
- Other statistics (later)

Collection approach: periodic, using sampling

Size Estimation Problem

```
S = SELECT *
     FROM   R1, …, Rn
     WHERE cond1 AND cond2 AND . . . AND condk
```

Given T(R1), T(R2), …, T(Rn)
Estimate T(S)

How can we do this ? Note: doesn't have to be exact.

Size Estimation Problem

```
S = SELECT *
      FROM    R1, …, Rn
      WHERE cond1 AND cond2 AND . . . AND condk
```

Remark: $T(S) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Selectivity Factor

- Each condition *cond* reduces the size by some factor called *selectivity factor*
- Assuming independence, multiply the selectivity factors

Example

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

What is the estimated size of the query output ?

Example

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

What is the estimated size of the query output ?

30k * 200k * 10k * 1/3 * 1/10 * ½
= 1TB

Statistical Model

What is the probability space?

```
S = SELECT list
      FROM    R_1 as x_1, …, R_k as x_k
      WHERE Cond  -- a conjunction of predicates
```

Statistical Model

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

(x_1, x_2, …, x_k), drawn randomly, independently from R_1, ..., R_k

Pr(R_1.A = 40) = prob. that random tuple in R_1 has A=40

Descriptive attribute

Join indicator (in class…)

Pr(R_1.A = 40 and $J_{R1.B = R2.C}$ and R_2.D = 90) = prob. that …

E[|SELECT ... WHERE Cond|] = Pr(Cond) * T(R_1) * T(R_2) * ... * T(R_k)

Statistical Model

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

Three simplifying assumptions

Uniform: $\Pr(R_1.A = \text{'a'}) = 1/V(R_1, A)$

Attribute Indep.: $\Pr(R_1.A = \text{'a' and } R_1.B = \text{'b'}) = \Pr(R_1.A = \text{'a'}) \Pr(R_1.B = \text{'b'})$

Join Indep.: $\Pr(R_1.A = \text{'a' and } J_{R1.B = R2.C}) = \Pr(R_1.A = \text{'a'}) \Pr(J_{R1.B = R2.C})$

Rule of Thumb

- If selectivities are unknown, then:
  selectivity factor = 1/10
  [System R, 1979]

Using Data Statistics

- Condition is A = c /* value selection on R */
  - Selectivity = 1/V(R,A)
- Condition is A < c /* range selection on R */
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))T(R)
- Condition is A = B /* $R \bowtie_{A=B} S$ */
  - Selectivity = 1 / max(V(R,A),V(S,A))
  - (will explain next)

Selectivity of Join Predicates

Assumption:

- *Containment of values*: if V(R,A) <= V(S,B), then the set of A values of R is included in the set of B values of S
  - Note: this indeed holds when A is a foreign key in R, and B is a key in S

Selectivity of Join Predicates

Assume $V(R,A) <= V(S,B)$

- Each tuple t in R joins with $T(S)/V(S,B)$ tuple(s) in S
- Hence $T(R \bowtie_{A=B} S) = T(R)\ T(S)\ /\ V(S,B)$

In general: $T(R \bowtie_{A=B} S) = T(R)\ T(S)\ /\ \max(V(R,A),V(S,B))$

Selectivity of Join Predicates

Example:

- $T(R) = 10000$, $T(S) = 20000$
- $V(R,A) = 100$, $V(S,B) = 200$
- How large is $R \bowtie_{A=B} S$?

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(<u>ssn</u>, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Estimate = 1200 Estimate = 1*80 + 5*500 = 2580

Types of Histograms

- How should we determine the bucket boundaries in a histogram ?

Types of Histograms

- How should we determine the bucket boundaries in a histogram ?

- Eq-Width
- Eq-Depth
- Compressed
- V-Optimal histograms

Employee(ssn, name, age)

Histograms

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Eq-depth:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	1800	2000	2100	2200	1900	1800

Compressed: store separately highly frequent values: (48,1900)

V-Optimal Histograms

- Defines bucket boundaries in an optimal way, to minimize the error over all point queries
- Computed rather expensively, using dynamic programming
- Modern databases systems use V-optimal histograms or some variations

Discussion in Class

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY ?
- *Not* updated during database update, but recomputed periodically
  - WHY ?

Multidimensional Histograms

Classical example:

SQL query:

```
SELECT …  FROM    …
WHERE  Person.city = 'Seattle' …
```

User "optimizes" it to:

```
SELECT …  FROM    …
WHERE  Person.city = 'Seattle'
       and Person.state = 'WA'
```

Big problem! (Why?)

Multidimensional Histograms

- Store distributions on two or more attributes
- Curse of dimensionality: space grows exponentially with dimension
- In practice: only two dimensional histograms

The New Hipster: NoSQL

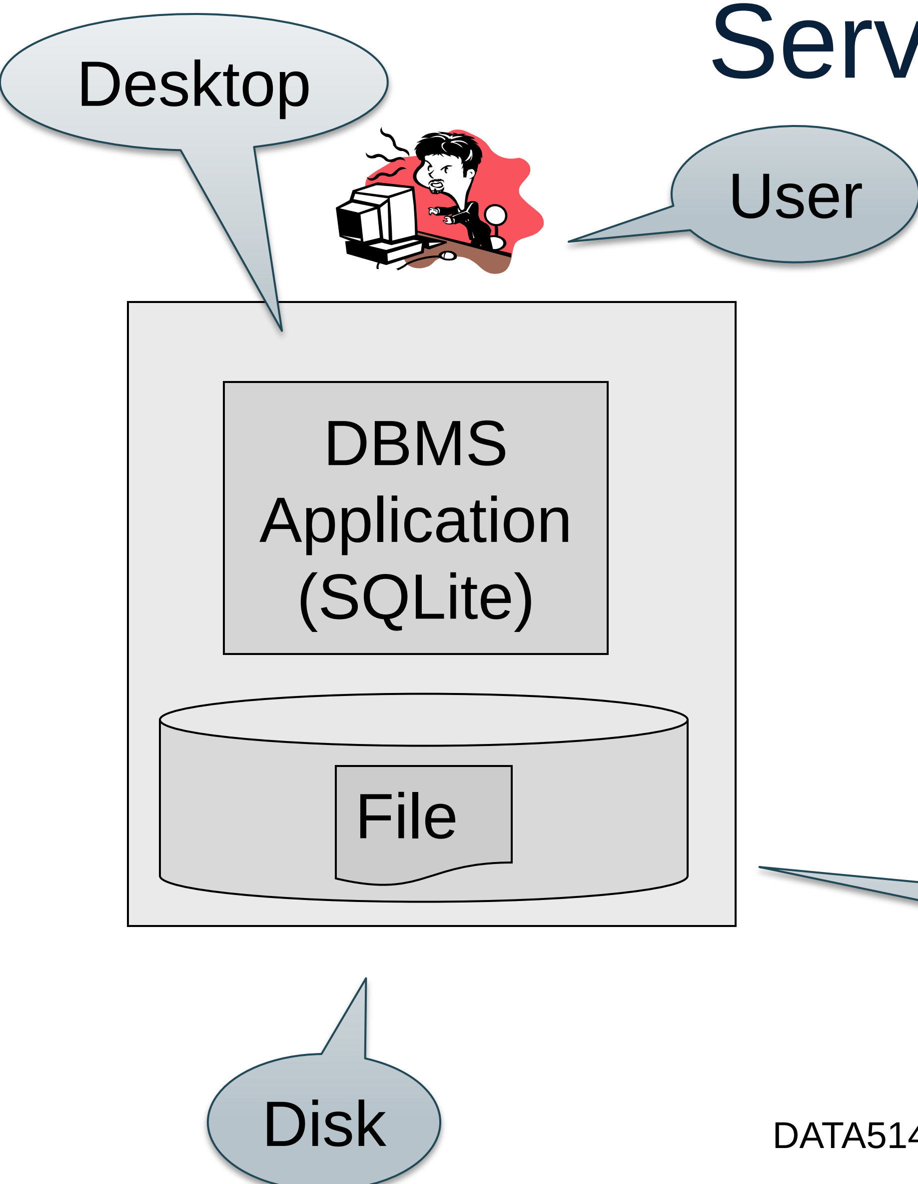
NoSQL Motivation

- Originally motivated by Web 2.0 applications
- Goal is to scale simple OLTP-style workloads to thousands or millions of users
  (in class: OLTP v.s. OLAP)
- Users are doing both updates and reads

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
  - Simpler transactions

Serverless

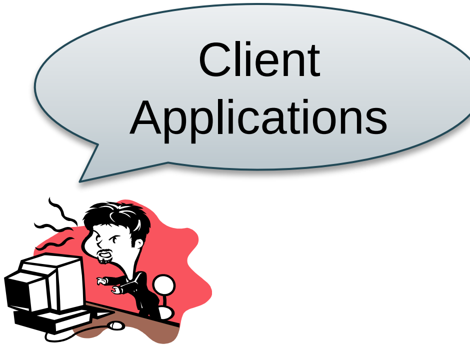


SQLite:

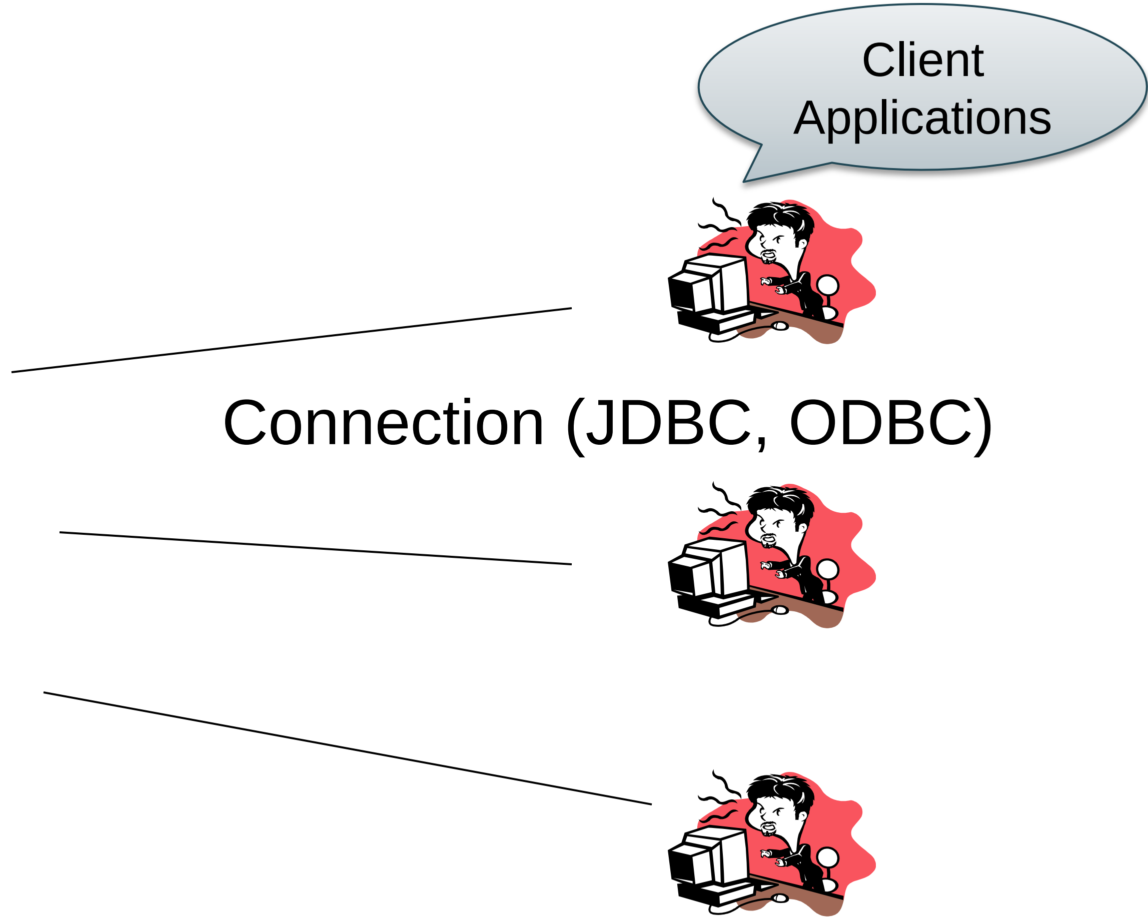
- One data file
- One user
- One DBMS application

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

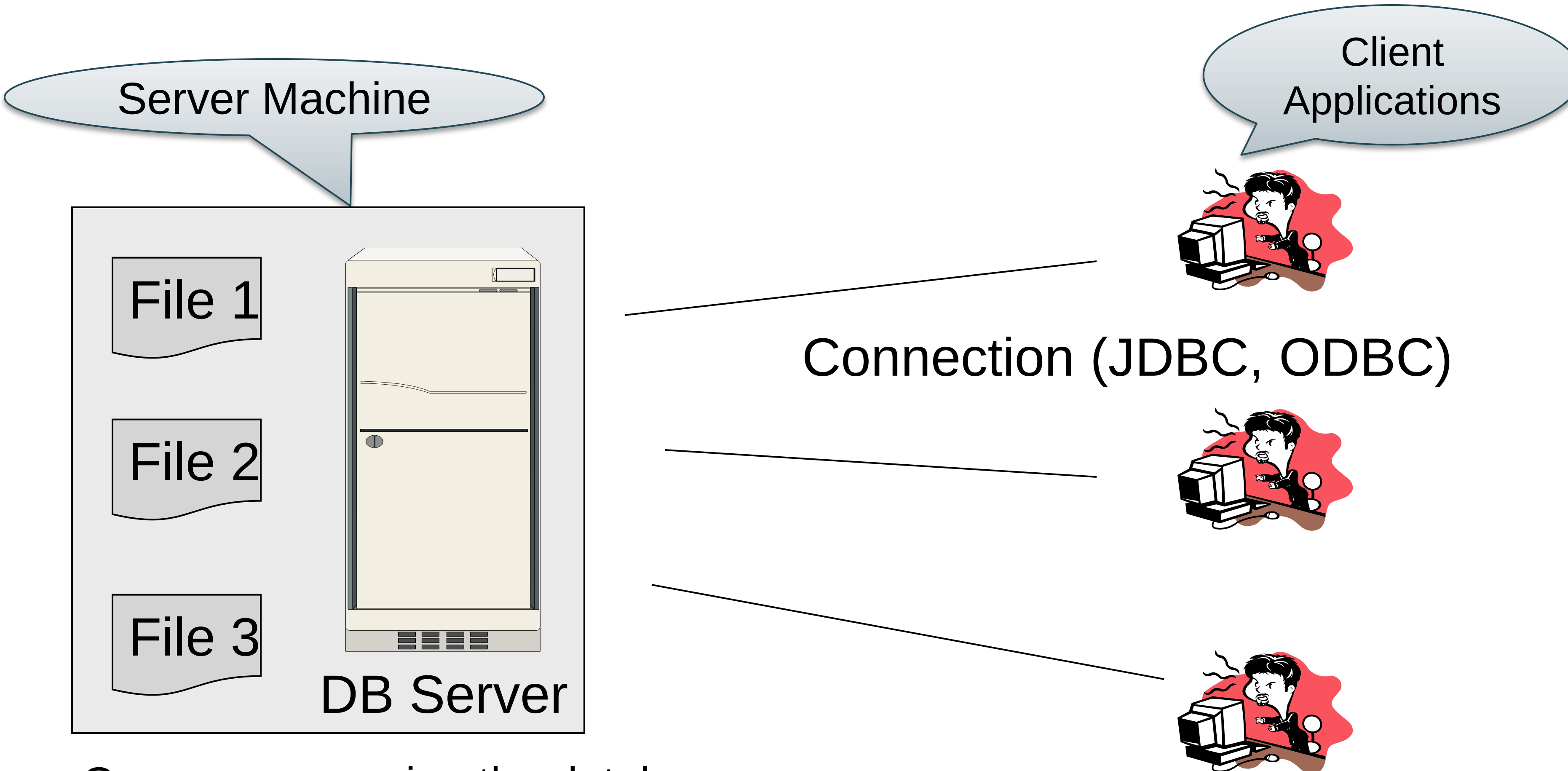
Client-Server



Client-Server

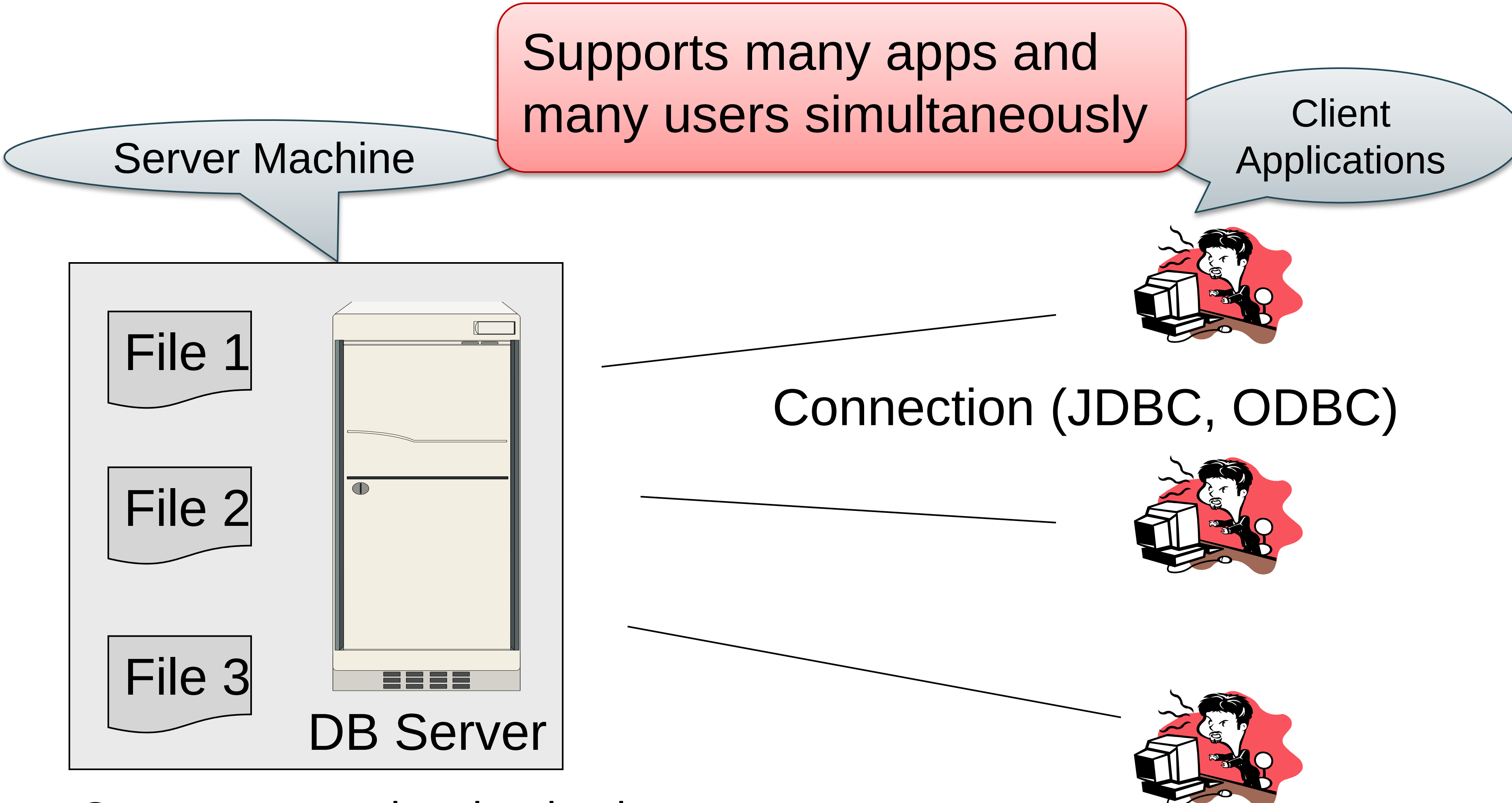


Client-Server



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

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)

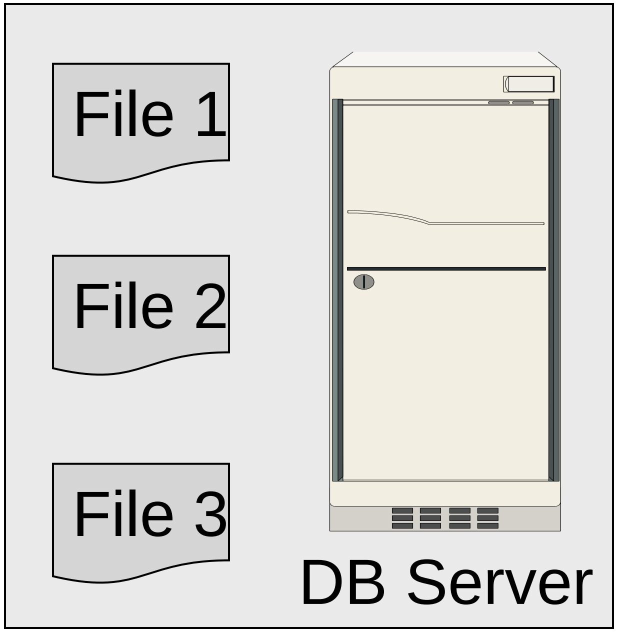
Client-Server

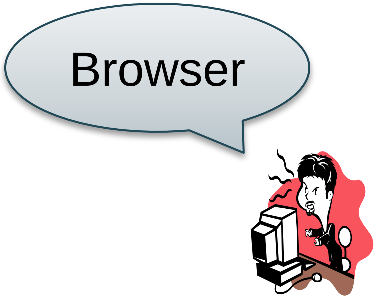
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft’s Management Studio (for SQL Server), or
  - psql (for postgres)
  - Some Java program or some C++ program

Client-Server

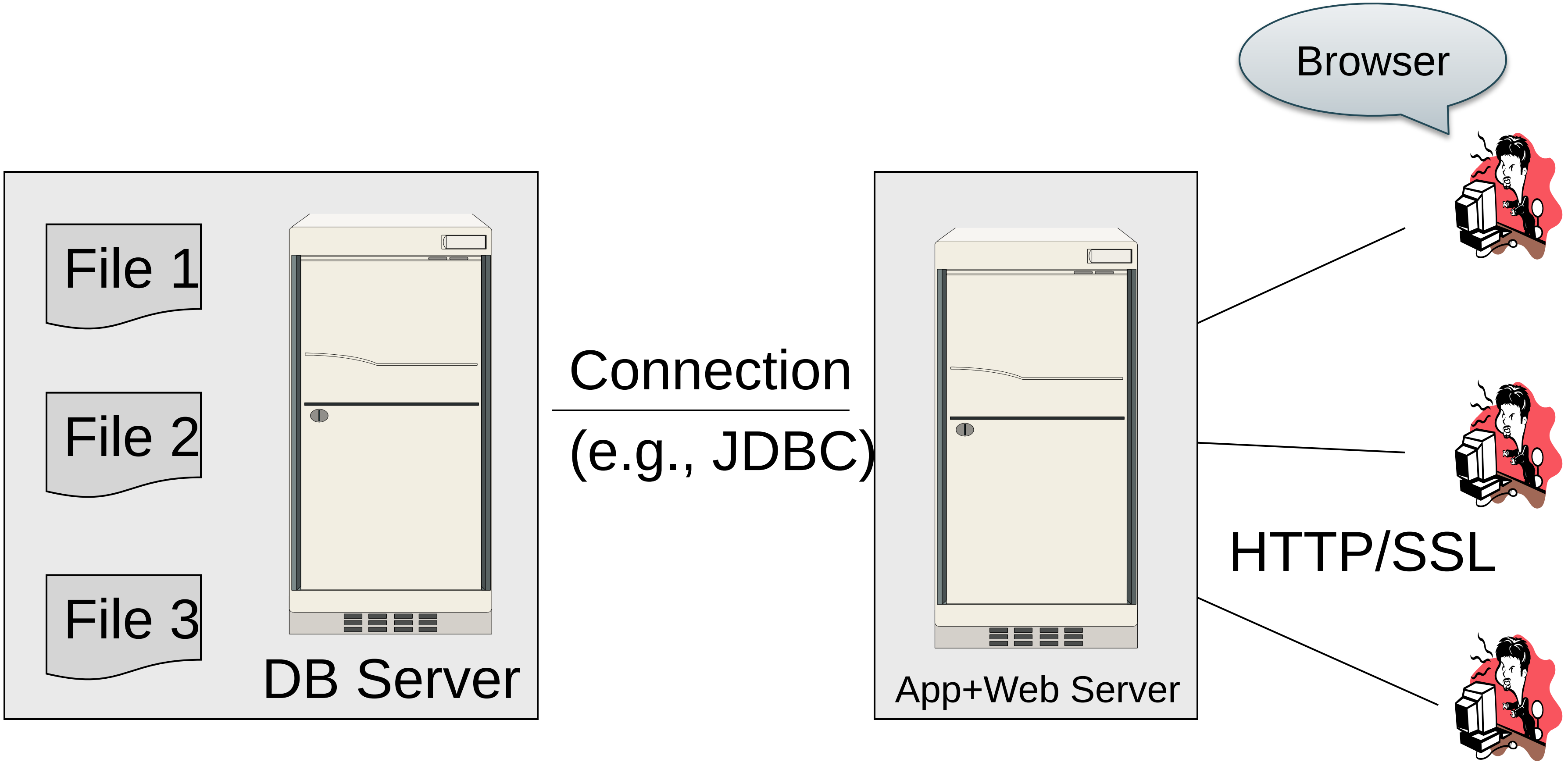
- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft's Management Studio (for SQL Server), or
  - psql (for postgres)
  - Some Java program (HW5) or some C++ program
- Clients "talk" to server using JDBC/ODBC protocol

3-Tiers DBMS Deployment

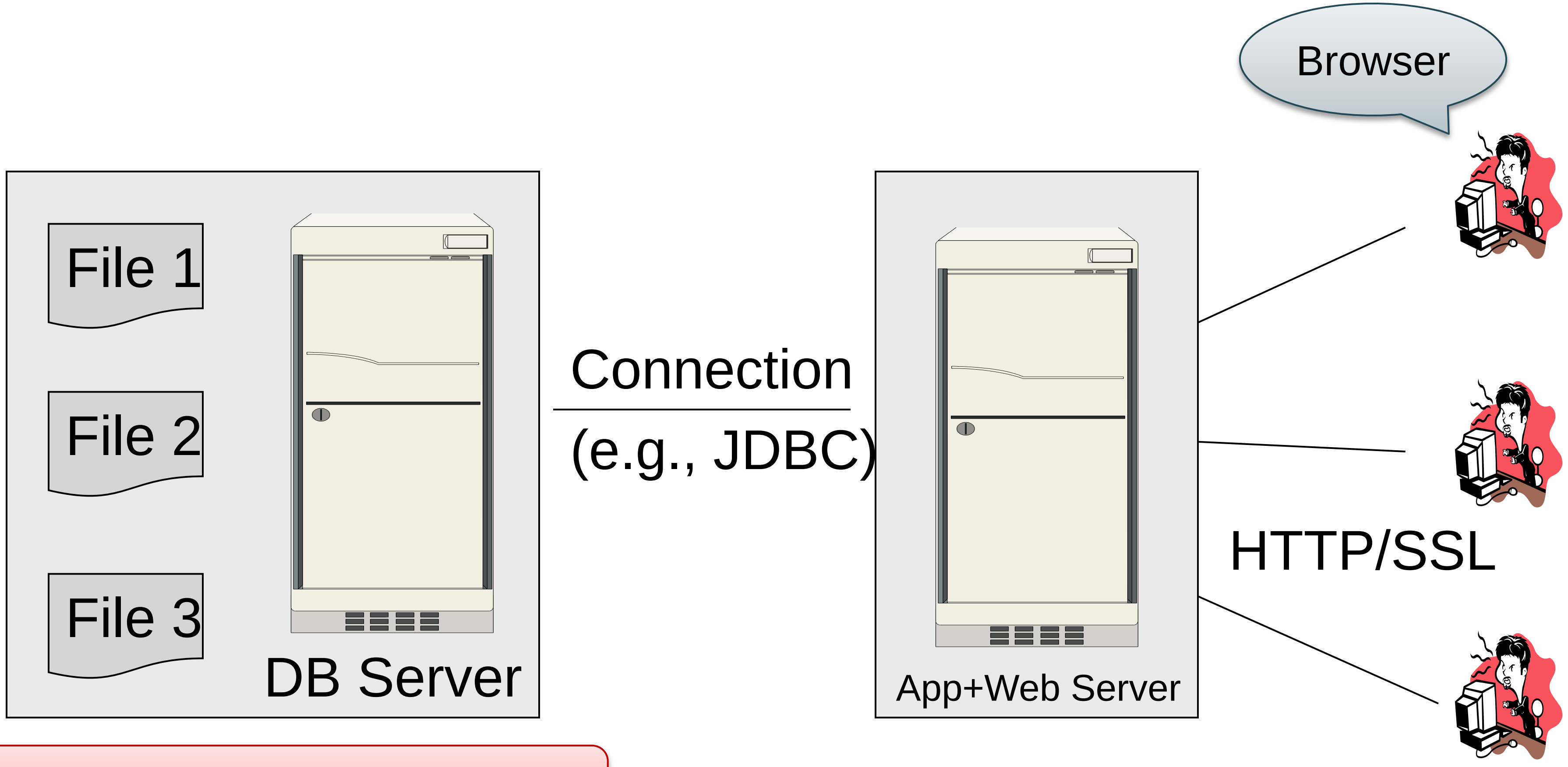




3-Tiers DBMS Deployment

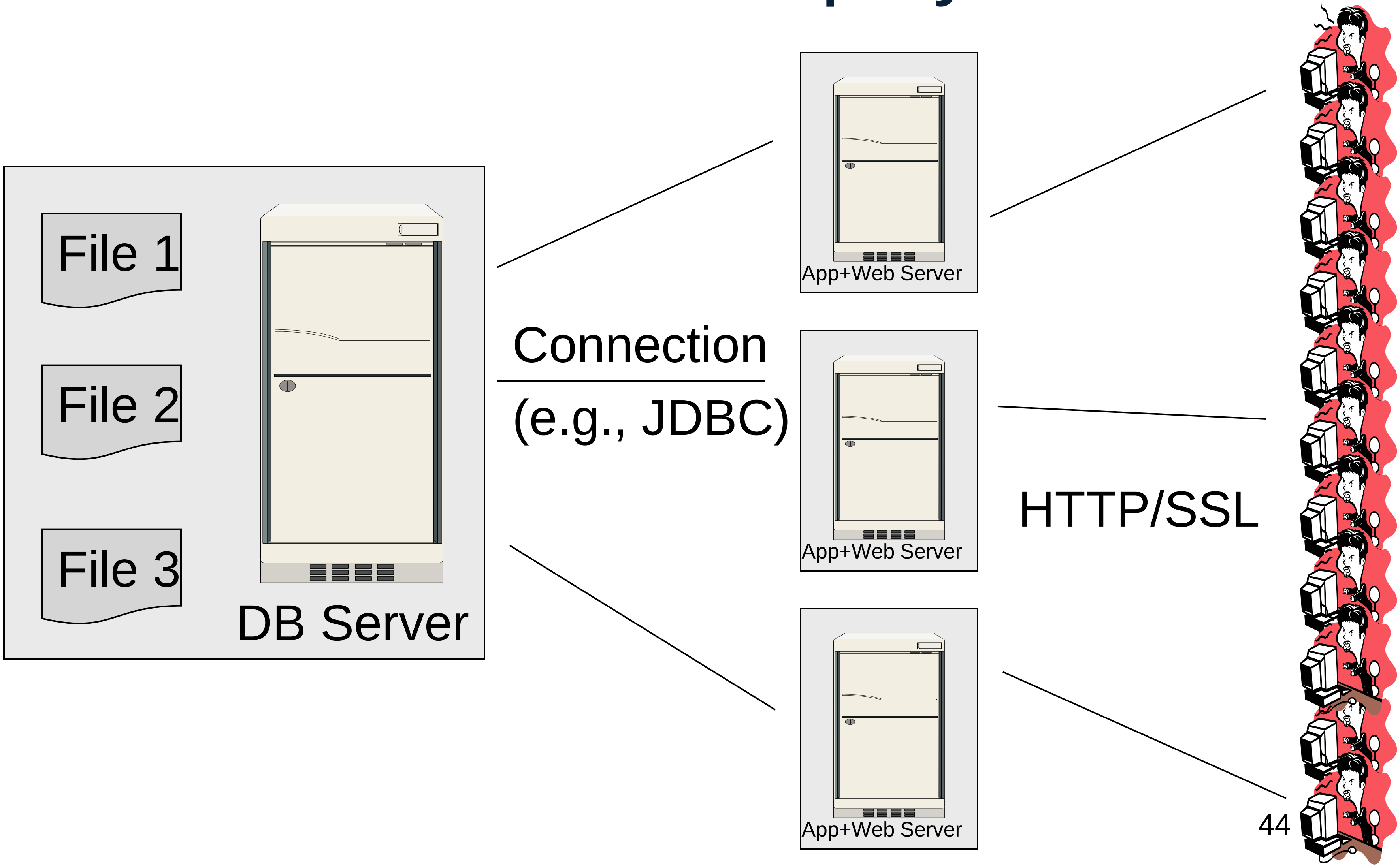


3-Tiers DBMS Deployment



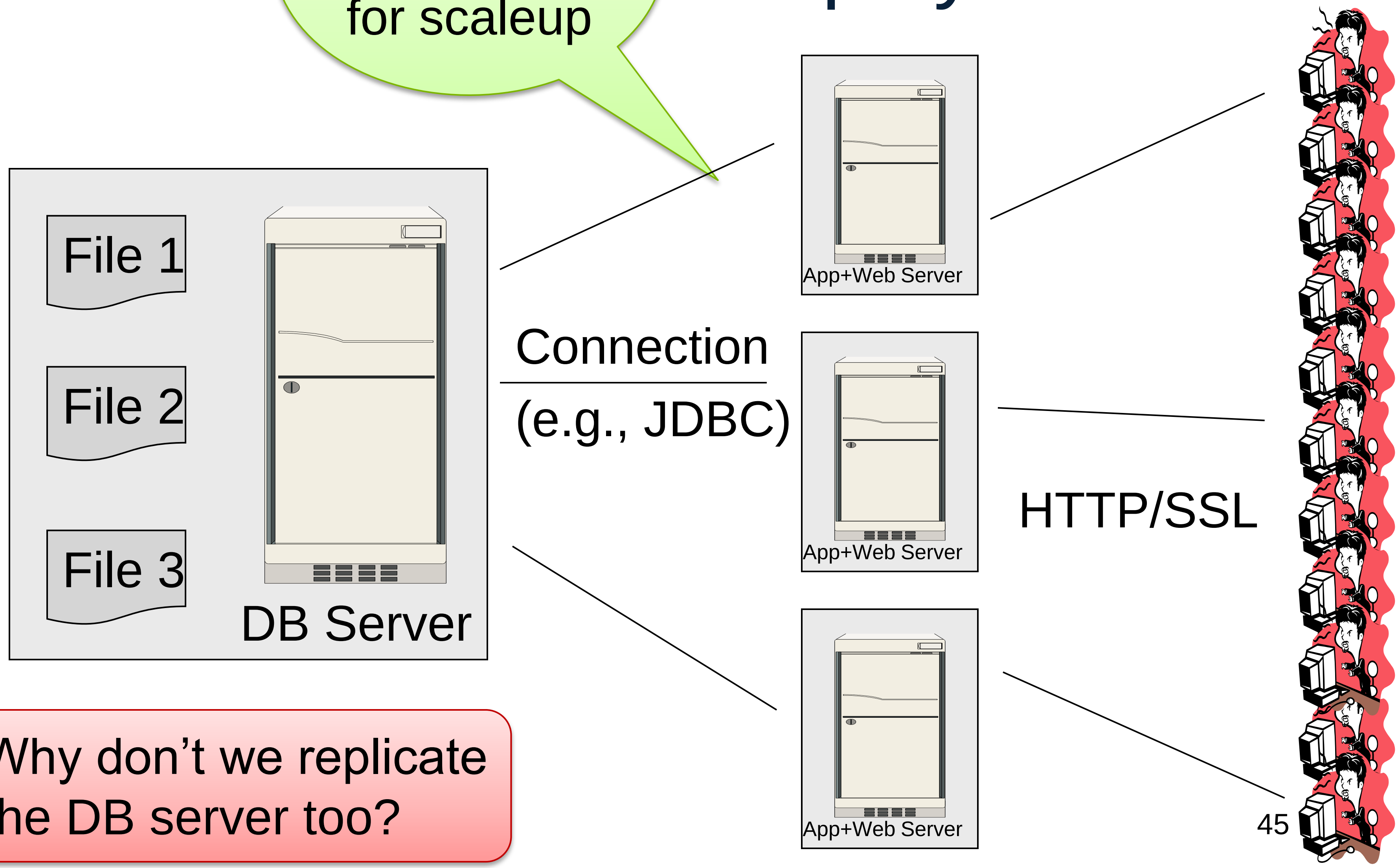
Web-based applications

3-Tiers DBMS Deployment



3-T … S Deployment

Replicate
App server
for scaleup



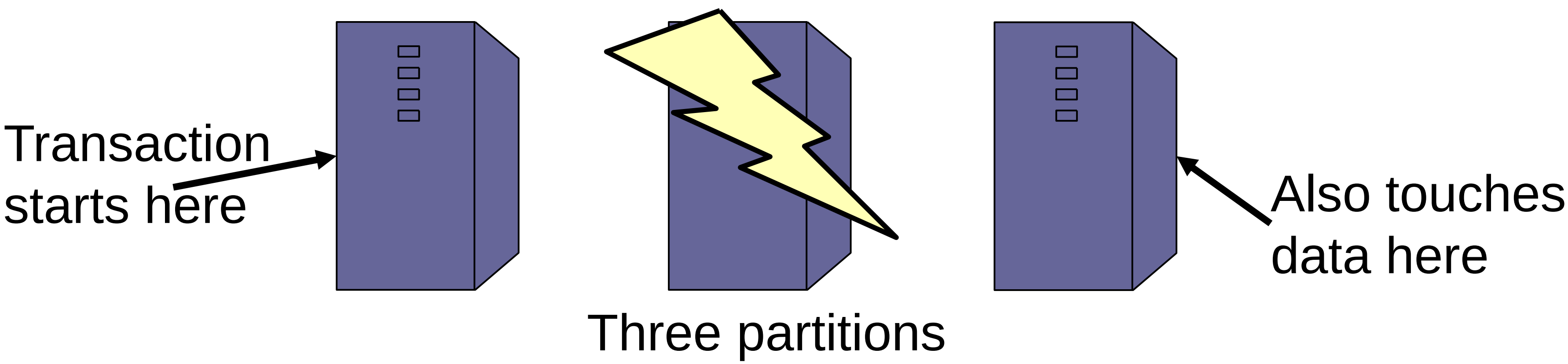
Why don’t we replicate the DB server too?

Replicating the Database

- Much harder, because the state must be unique, in other words the database must act as a whole
- Two basic approaches:
  - Scale up through partitioning
  - Scale up through replication

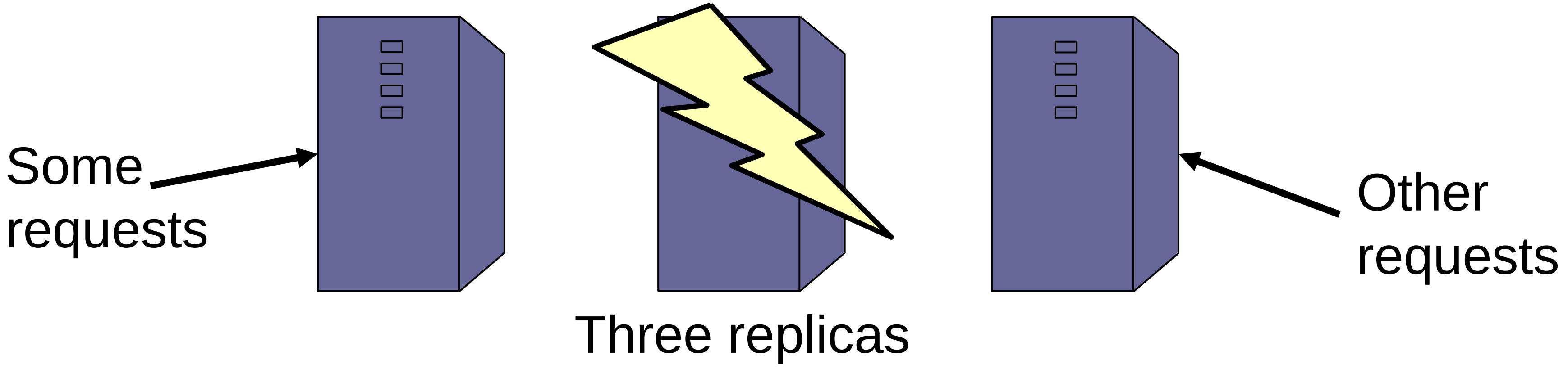
Scale Through Partitioning

- 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!



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

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**
  - 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?

- 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

– Data remains in main memory

– One type of impl.: distributed hash table

– Most systems also offer a persistence option

– Others use replication to provide fault-tolerance

– Some offer ACID transactions others do not

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

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
  - Limited, non-standard query language on JSon
- **Distribution / Partitioning**
  - Entire documents, as for key/value pairs

We will discuss JSon today

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
- HBase is an open source implementation of BigTable

JSon and Semistructured Data

Where We Are

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

- Relational data model
  - 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

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 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”
}
```

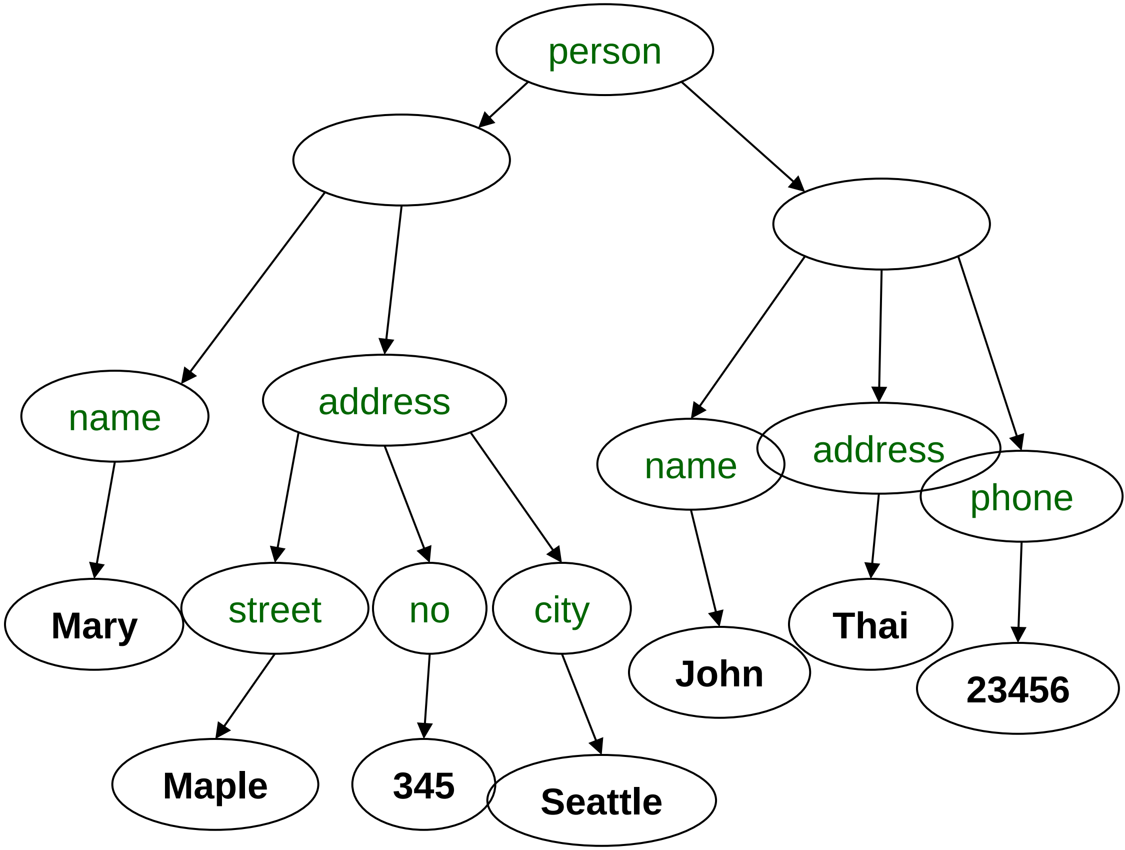
```
{"id":"07",
   ”title": ”Databases”,
   "author": [”Garcia-Molina”,
              ”Ullman”,
              ”Widom”]
}
```

JSon Datatypes

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

JSon Semantics: a Tree !

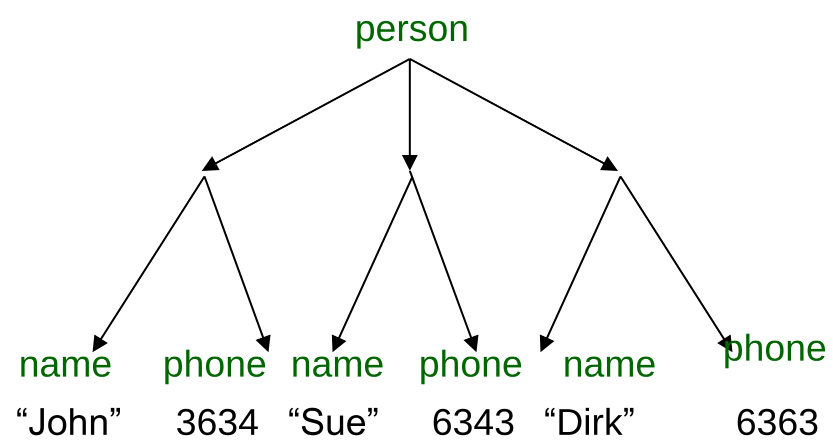
```
{"person":
  [ {"name": "Mary",
     "address":
       {"street":"Maple",
        "no":345,
        "city": "Seattle"}},
    {"name": "John",
     "address": "Thailand",
     "phone":2345678}}
  ]
}
```



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