Database Management Systems
CSEP 544

Lecture 4: Datalog and NoSQL

Announcements

- HW3 due today

- HW4 posted
  - Please start early!

- Today:
  - Datalog (relational data model)
  - Non-relational data models

What is Datalog?

- Another *declarative* query language for relational model
  - Designed in the 80’s
  - Minimal syntax
  - Simple, concise, elegant
  - Extends relational queries with *recursion*

- Today:
  - Adopted by some companies for data analytics, e.g., LogicBlox (HW4)
  - Usage beyond databases: e.g., network protocols, static program analysis

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x),
             Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie
Intensional Database Predicates = IDB = Q1, Q2, Q3

Datalog: Terminology

head — body

atom — atom — atom (aka subgoal)

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
```

f, l = head variables

x,y,z = existential variables

In this class we discuss datalog evaluated under **set semantics**

More Datalog Terminology

Q(args) :- R1(args), R2(args), ...

Your book uses:
Q(args) :- R1(args) AND R2(args) AND ....

- $R_i(args_i)$ is called an atom, or a relational predicate
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, 'Douglas', 'Fowley') is true
- In addition to relational predicates, we can also have arithmetic predicates
  - Example: z > '1940'.
- Note: Logicblox uses <- instead of :-

Q(args) <- R1(args), R2(args), ...

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

```
Q1(y) :- Movie(x,y,z), z='1940'.
```

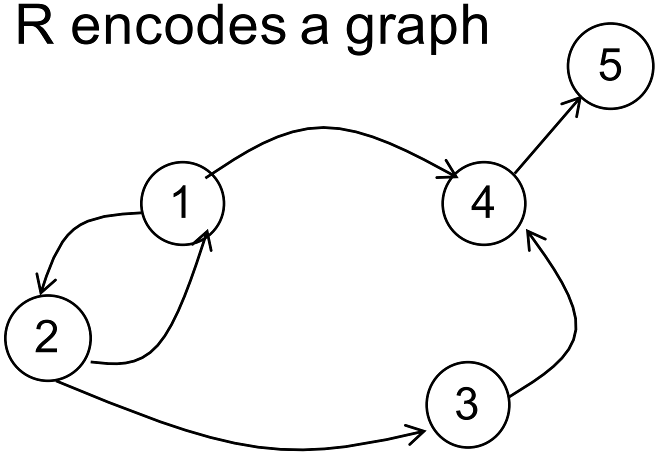
- For all values of x, y, z:
  if (x,y,z) is in the Movies relation, and that z = '1940' then y is in Q1 (i.e., it is part of the answer)
- Logically equivalent:
  $\forall\ y.\ [(\exists x.\ \exists\ z.\ \text{Movie}(x,y,z) \text{ and } z=\text{'1940'}) \Rightarrow \text{Q1}(y)]$
- That's why non-head variables are called "existential variables"
- We want the *smallest* set Q1 with this property (why?)

Datalog program

- A datalog program consists of several rules
- Importantly, rules may be recursive!
- Usually there is one distinguished predicate that's the output
- We will show an example first, then give the general semantics.

Example

R encodes a graph

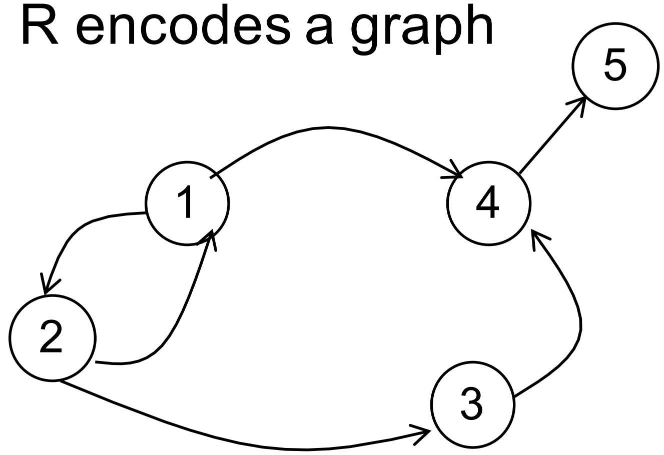


R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

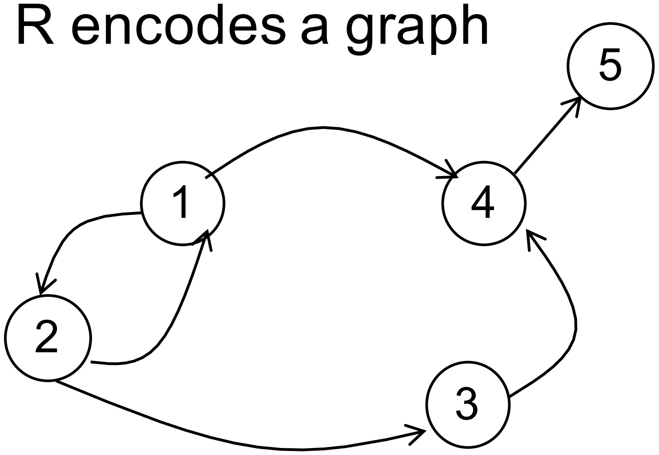
What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph




```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

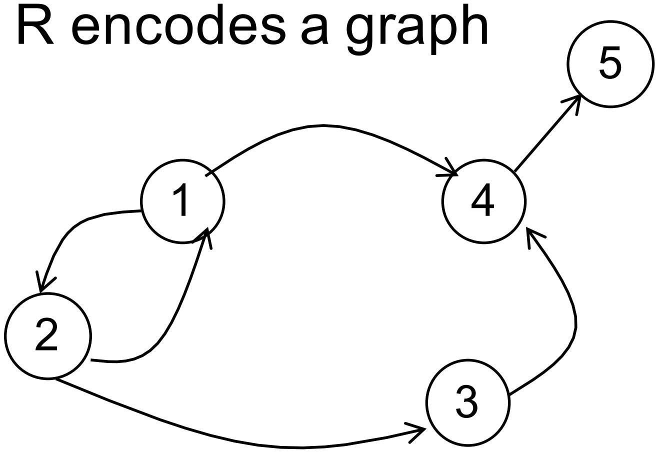
Initially:
T is empty.

Example

$T(x,y) \text{ :- } R(x,y)$
$T(x,y) \text{ :- } R(x,z),\ T(z,y)$

What does it compute?

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

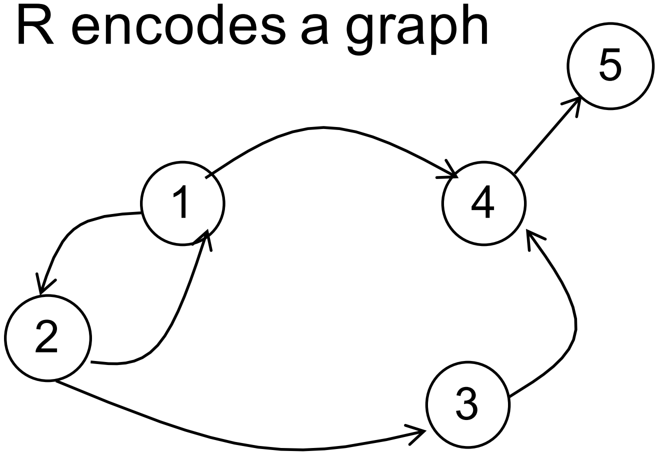
1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule generates nothing (because T is empty)

Example

R encodes a graph



```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

First rule generates this (rows 1–6)

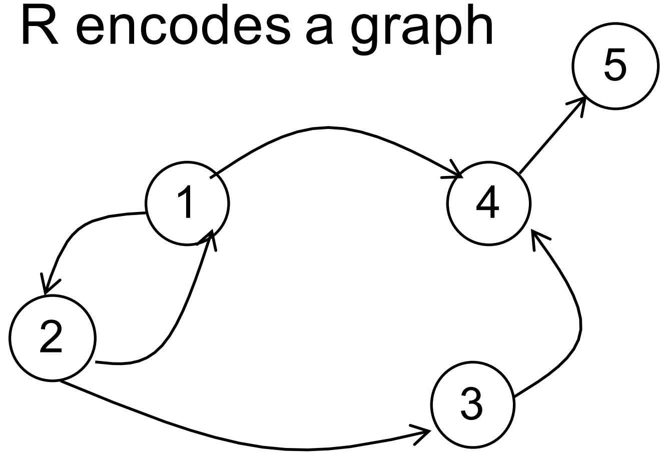
Second rule generates this (rows 7–12)

New facts

T(1,1) :- R(1,2), T(2,1)
x,y x z z y

Example

R encodes a graph



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

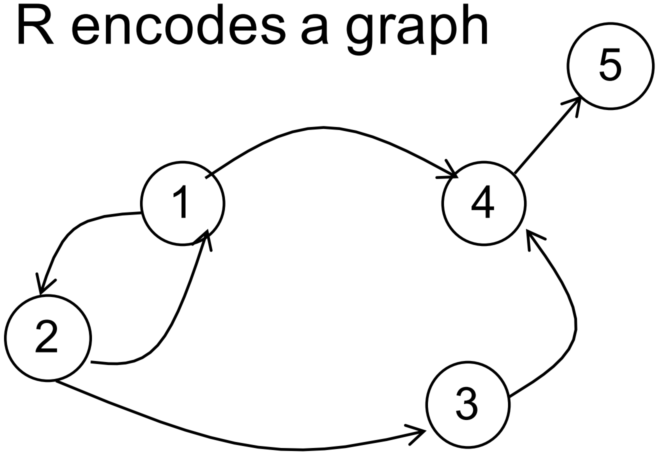
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

1	2	Both rules
2	1	
2	3	First rule
1	4	
3	4	
4	5	
1	1	
2	2	Second rule
1	3	
2	4	
1	5	
3	5	
2	5	New fact

Example

R encodes a graph



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

Fourth iteration
T =
(same)

No new facts.
DONE

This is called the **fixpoint semantics** of a datalog program

Demo

Evaluation of Datalog

How to evaluate a datalog program?

- Start:
  for every IDB D_i , $D_i^0 = \varnothing$
  $t = 0$

- Repeat:
  for every IDB D_i^{t+1} = eval rules(EDB, IDB_1^t, IDB_2^t, …)
  $t = t+1$

- Until:
  for every IDB $D_i^t = D_i^{t-1}$ (aka fixpoint)

- The answer is in D_1^t, D_2^t, …
- This is called **naive evaluation**.

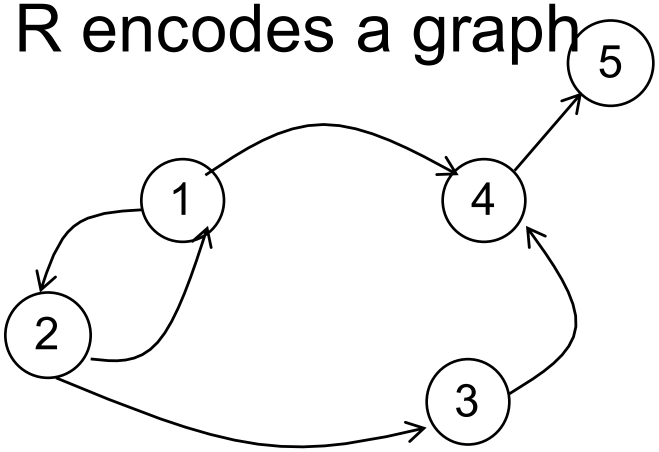
Evaluation of Datalog

- A datalog program w/o functions (+, *, ...) always terminates.
  – Hint: since the rules are monotone, hence:
    $\emptyset = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq ...$

- How many iterations of naive evaluation are needed before reaching fixpoint?

Three Equivalent Programs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

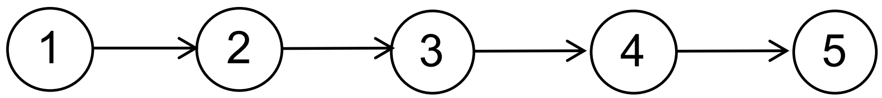
Left linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)
```

Non-linear

Question: which terminates in fewest iterations?

Three Equivalent Programs



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

R=

1	2
2	3
3	4
4	5

t = 0:

T=

1	2
2	3
3	4
4	5

t = 1:

T=

1	2
2	3
3	4
4	5
1	3
2	4
3	5

Second rule (rows 1 3, 2 4, 3 5)

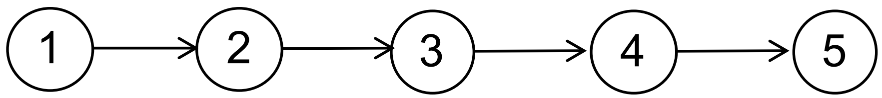
t = 2:

T=

1	2
2	3
3	4
4	5
1	3
2	4
3	5
1	4
1	5
2	5

Second rule (rows 1 4, 1 5, 2 5)

Three Equivalent Programs



T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

R=

1	2
2	3
3	4
4	5

t = 0:

T=

1	2
2	3
3	4
4	5

t = 1:

T=

1	2
2	3
3	4
4	5
1	3
2	4
3	5

Second rule (rows 1–3, 2–4, 3–5)

t = 2:

T=

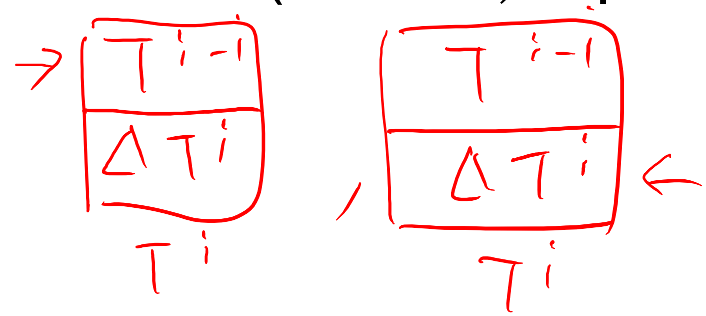
1	2
2	3
3	4
4	5
1	3
2	4
3	5
1	4
1	5
2	5

Second rule "rediscovered facts" (rows 1–3, 2–4, 3–5)

Second rule (rows 1–4, 1–5, 2–5)

Evaluation of Datalog

Idea: split a relation into “old” and “new” (aka “Δ”) tuples



T^{i+1} :- $Q(T^i, T^i)$

$= Q(T^{i-1} \cup \Delta T^i, T^{i-1} \cup \Delta T^i)$

$= Q(T^i \cup T^i) \cup Q(T^{i-1}, \Delta T^i) \cup Q(\Delta T^i, T^{i-1}) \cup Q(\Delta T^i \cup \Delta T^i)$

$= T^i \cup Q(T^{i-1}, \Delta T^i) \cup Q(\Delta T^i, T^{i-1}) \cup Q(\Delta T^i \cup \Delta T^i)$

- Now we can evaluate on smaller relations
  - But need to keep track of the Δ tuples
- This is the basis of incremental query processing

Evaluation of Datalog

- Start:

  for every IDB D_i , $D_i^0 = \varnothing$

  for every IDB ΔD_i^1 = eval rules(EDB, IDB_1^0, IDB_2^0, …)

  t = 0

- Repeat:

  for every IDB $D_i^t = D_i^{t-1} \cup \Delta D_i^t$

  for every IDB ΔD_i^1 = eval rules(EDB, IDB_1^t, IDB_2^t, …)

  and compute Δ for each IDB

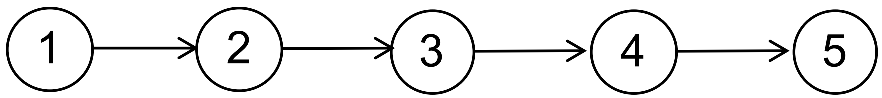
  t = t+1

- Until:

  for every IDB $\Delta D_i^t = \varnothing$ (aka fixpoint)

- The answer is in D_1^t, D_2^t, …
- This is called the **semi-naive** evaluation of Datalog

Semi-Naive Evaluation



T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

R=

1	2
2	3
3	4
4	5

T^0=

1	2
2	3
3	4
4	5

T^1=

1	2	T^0
2	3	
3	4	
4	5	
1	3	ΔT^1
2	4	
3	5	

T^2=

1	2	T^1
2	3	
3	4	
4	5	
1	3	
2	4	
3	5	
1	4	$Q(T^0, \Delta T^1)$
1	5	$Q(\Delta T^1, \Delta T^1)$
2	5	$Q(T^0, \Delta T^1)$

T^2 :- T^1 U $Q(T^0, \Delta T^1)$ U $Q(\Delta T^1, T^0)$ U $Q(\Delta T^1$ U $\Delta T^1)$

Extensions

- Functional data model (LogicBlox)
- Aggregates, negation
- Stratified datalog

Functional Data Model

- Relational data model:
  `Person(Alice, Smith) = true`
  `Person(Bob, Peters) = false`

First	Last
Alice	Smith
Bob	Toth
Carol	Unger

- Functional data model:
  `Person[Alice,Smith]` = some value v
- This is just a syntactic sugar for relations with keys

Functional Data Model

- `Person(first,last,friends)` (note the key)

first	last	friends
Alice	Smith	22
Bob	Toth	5
Carol	Unger	9

- Functional model:

```
Person[Alice,Smith]=22
Person[Bob,Toth]=5
Person[Carol,Unger]=9
```

first	last	
Alice	Smith	=22
Bob	Toth	=5
Carol	Unger	=9

Aggregates

Count the number of tuples in p and store the result in count_p

```
count_p[]=v <- agg<<v=count()>> p(_)
```

Meaning (in SQL)

```
select   count(*) as v
from     p
```

Aggregates

General syntax in Logicblox:

```
Q[headVars]=v <- agg<<v=AGG_NAME(w)>> R1(x₁),R2(x₂),...
```

Meaning (in SQL)

```
select   headVars, AGG_NAME(w) as v
from     R1, R2, ...
where    ...
group by headVars
```

ParentChild(p,c)

Example

For each person, compute the total number of descendants

```
/* We use Logicblox syntax (as in the homework) */
```

ParentChild(p,c)

Example

For each person, compute the total number of descendants

```
/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
```

ParentChild(p,c)

Example

For each person, compute the total number of descendants

```
/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
/* For each person, count the number of descendants */
N[x] = m  <-  agg<<m = count()>> D(x,y).
```

ParentChild(p,c)

Example

For each person, compute the total number of descendants

```
/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
/* For each person, count the number of descendants */
N[x] = m  <-  agg<<m = count()>> D(x,y).
/* Find the number of descendants of Alice */
Q(d)  <-  N["Alice"]=d.
```

Negation: use !

Find all descendants of Alice,
who are not descendants of Bob

```
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
/* Compute the answer: notice the negation */
Q(x)  <-  D("Alice",x), !D("Bob",x).
```

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

U1(x,y) :- ParentChild("Alice",x), y != "Bob"

U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)

A datalog rule is *safe* if every variable appears in some positive relational atom

Safe Datalog Rules

- Recursion does not cope well with aggregates or negation
- Example: what does this mean?

```
A() <- !B().
B() <- !A().
```

- Can't evaluate using naive / semi-naive algorithm!

Stratified Datalog

- A datalog program is *stratified* if it can be partitioned into strata s.t., for all n, only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.

- I.e., the program can be divided such that all variables have appeared in the head of some rule before they are used negatively / in an aggregate.

- LogicBlox accepts only stratified datalog.

Stratified Datalog

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
N[x] = m  <-  agg<<m = count()>> D(x,y).
Q(d)  <-  N["Alice"]=d.
```

Stratum 1

Stratum 2

May use D in an agg because was defined in previous stratum

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
Q(x)  <-  D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use !D

```
A() <- !B().
B() <- !A().
```

Non-stratified

Cannot use !A

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Union:

R(A,B,C) $\cup$ S(D,E,F)

U(x,y,z) :- R(x,y,z)

U(x,y,z) :- S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Intersection:

$R(A,B,C) \cap S(D,E,F)$

I(x,y,z) :- R(x,y,z), S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Selection: $\sigma_{x>100 \text{ and } y=\text{'foo'}}(R)$

L(x,y,z) :- R(x,y,z), x > 100, y='foo'

Selection: $\sigma_{x>100 \textbf{ or } y=\text{'foo'}}(R)$

L(x,y,z) :- R(x,y,z), x > 100

L(x,y,z) :- R(x,y,z), y='foo'

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Equi-join: $R \bowtie_{R.A=S.D \text{ and } R.B=S.E} S$

J(x,y,z,q) :- R(x,y,z), S(x,y,q)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Projection:

P(x) :- R(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

To express difference, we add negation

!

D(x,y,z) :- R(x,y,z), NOT S(x,y,z)

Examples

R(A,B,C)

S(D,E,F)

T(G,H)

Translate: $\pi_A(\sigma_{B=3}(R))$

A(a) :- R(a,3,_)

Underscore used to denote an "anonymous variable"

Each such variable is unique

Examples

R(A,B,C)

S(D,E,F)

T(G,H)

Translate: $\pi_A(\sigma_{B=3}(R) \bowtie_{R.A=S.D} \sigma_{E=5}(S))$

A(a) :- R(a,3,_), S(a,5,_)

These are different "_"s

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find Joe's friends, and Joe's friends of friends.

```
A(x) :- Friend('Joe', x)
A(x) :- Friend('Joe', z), Friend(z, x)
```

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find all of Joe's friends who do not have any friends except for Joe:

```
JoeFriends(x) :- Friend('Joe',x)
NonAns(x) :- JoeFriends(x), Friend(x,y), y != ‘Joe’
A(x) :- JoeFriends(x), NOT NonAns(x)
```

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find all people such that all their enemies' enemies are their friends

- Q: if someone doesn't have any enemies nor friends, do we want them in the answer?
- A: Yes!

```
Everyone(x) :- Friend(x,y)
Everyone(x) :- Friend(y,x)
Everyone(x) :- Enemy(x,y)
Everyone(x) :- Enemy(y,x)
NonAns(x) :- Enemy(x,y),Enemy(y,z), NOT Friend(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

Friend(name1, name2)
Enemy(name1, name2)

More Examples

Find all persons x that have a friend all of whose enemies are x's enemies.

```
Everyone(x) :- Friend(x,y)
NonAns(x) :- Friend(x,y) Enemy(y,z), NOT Enemy(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

Datalog Summary

- EDB (base relations) and IDB (derived relations)
- Datalog program = set of rules
- Datalog is recursive

- Some reminders about semantics:
  - Multiple atoms in a rule mean join (or intersection)
  - Variables with the same name are join variables
  - Multiple rules with same head mean union

Relational Data Model

- Data is stored in flat relations
- Physical and data independence

- Three languages for data manipulation:
  - SQL: declarative
  - Relational algebra: imperative
  - Datalog: declarative / logical

  - Each has advantages and disadvantages

NoSQL

Class overview

- Data models
  - Relational: SQL, RA, and Datalog
  - NoSQL: SQL++
- RDMBS internals
  - Query processing and optimization
  - Physical design
- Parallel query processing
  - Spark and Hadoop
- Conceptual design
  - E/R diagrams
  - Schema normalization
- Transactions
  - Locking and schedules
  - Writing DB applications

Data models

Query Processing

Using DBMS

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

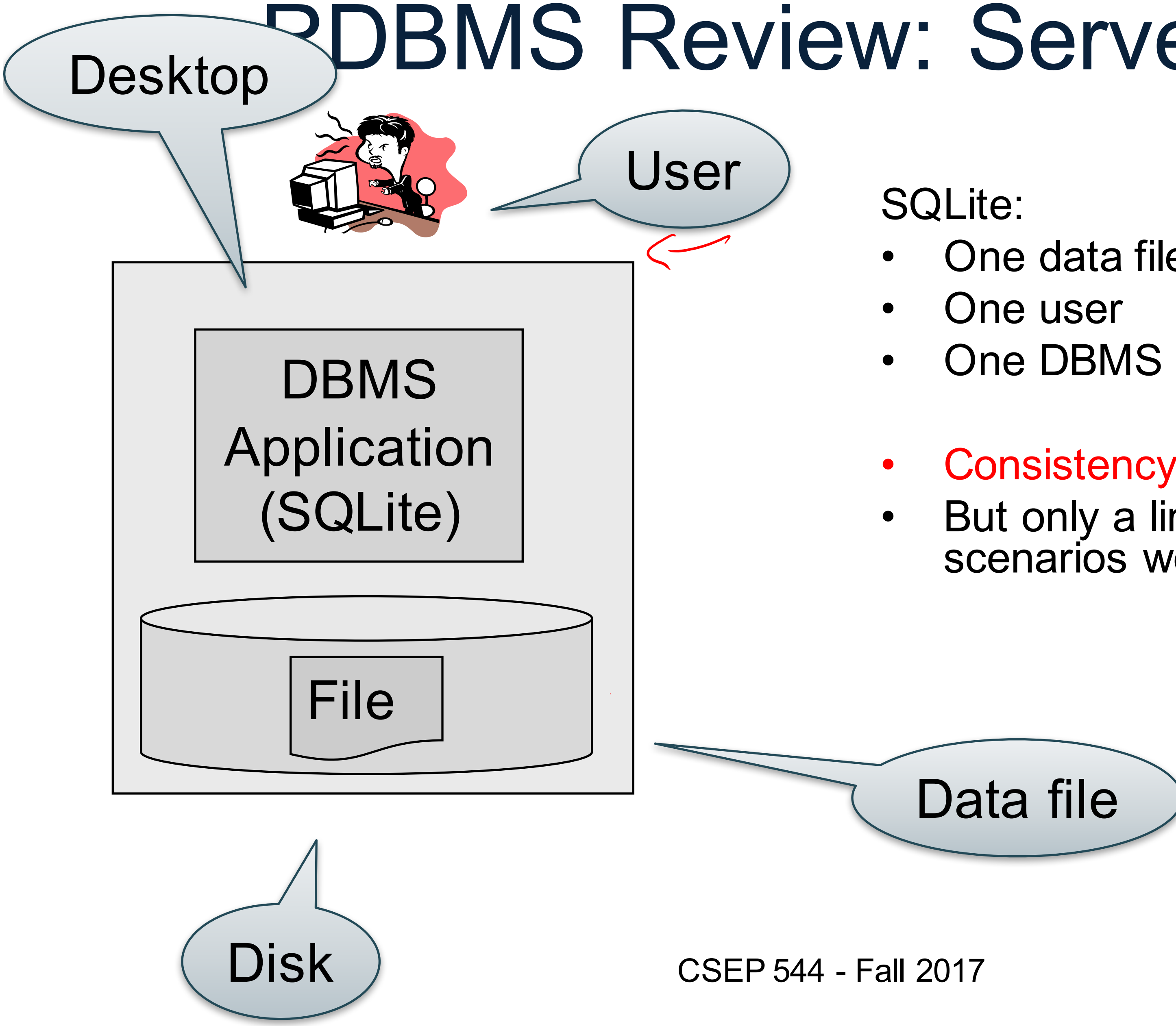
NoSQL Motivation

- 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

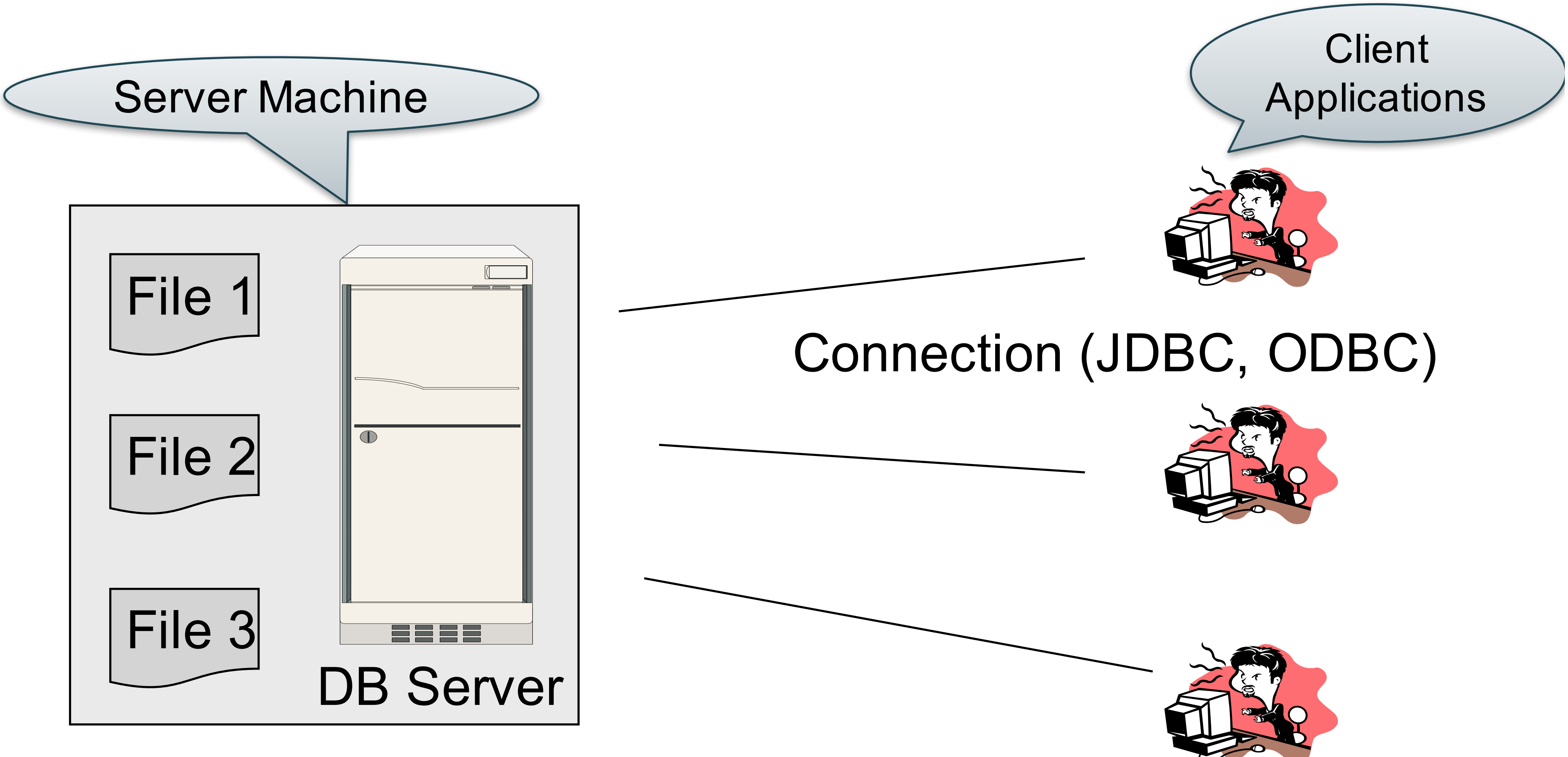


SQLite:

- One data file
- One user
- One DBMS application

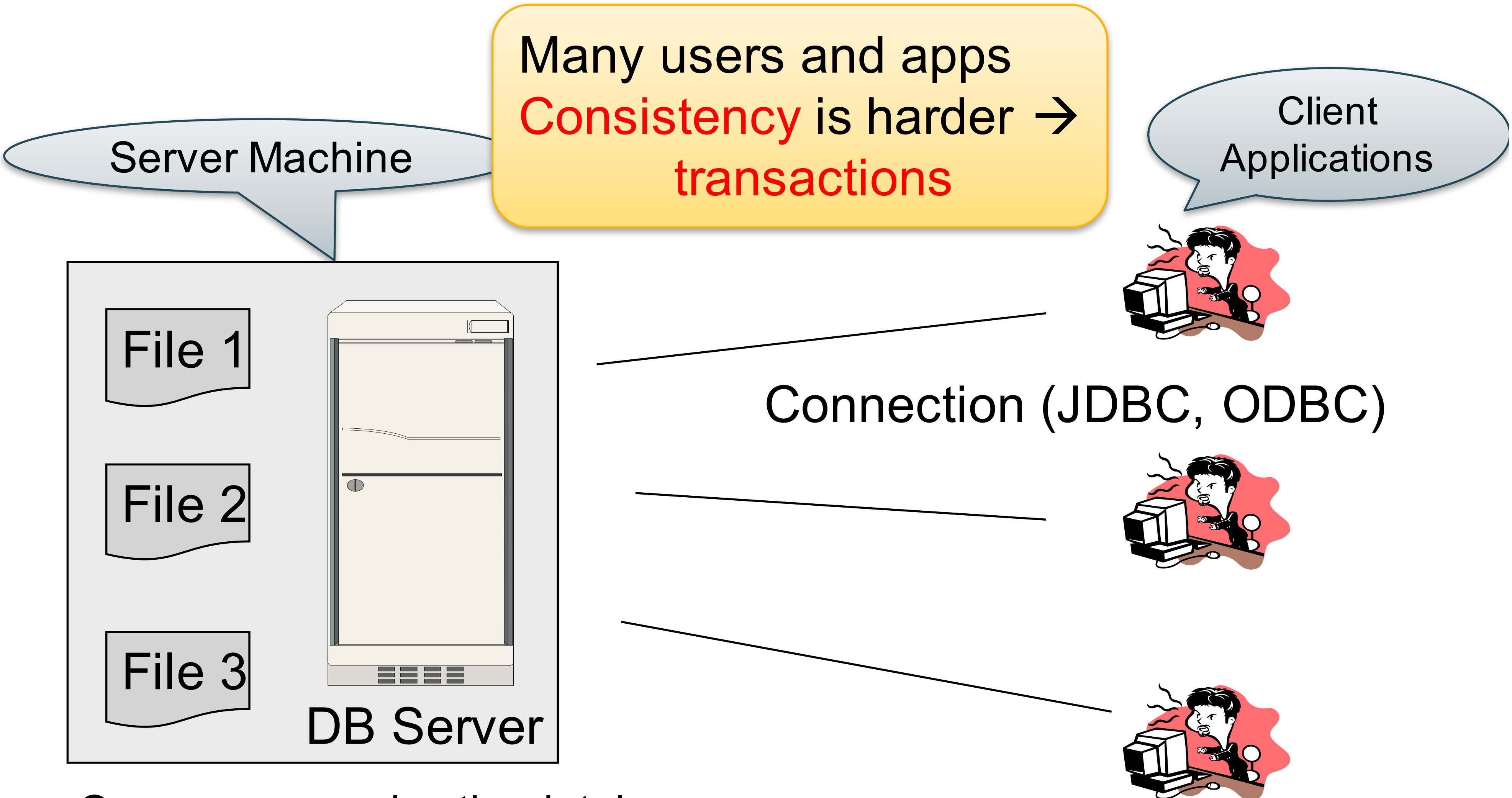
- Consistency is easy
- But only a limited number of scenarios work with such model

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)

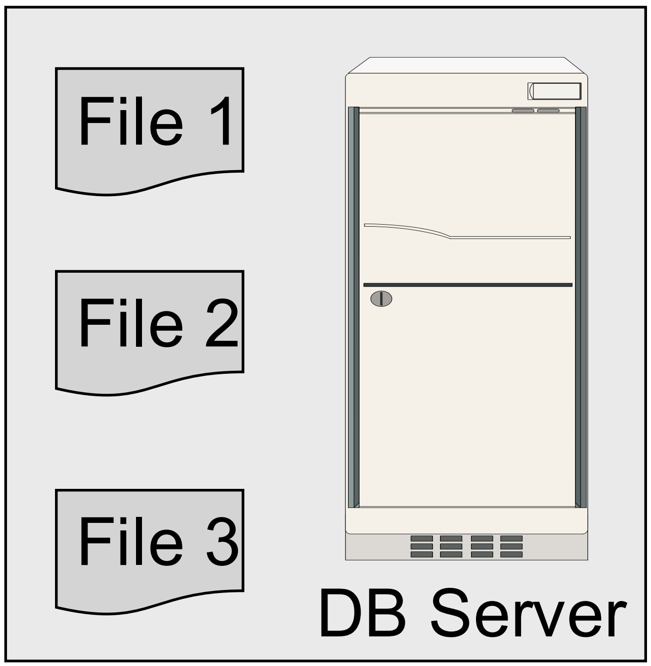
Client-Server

- 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

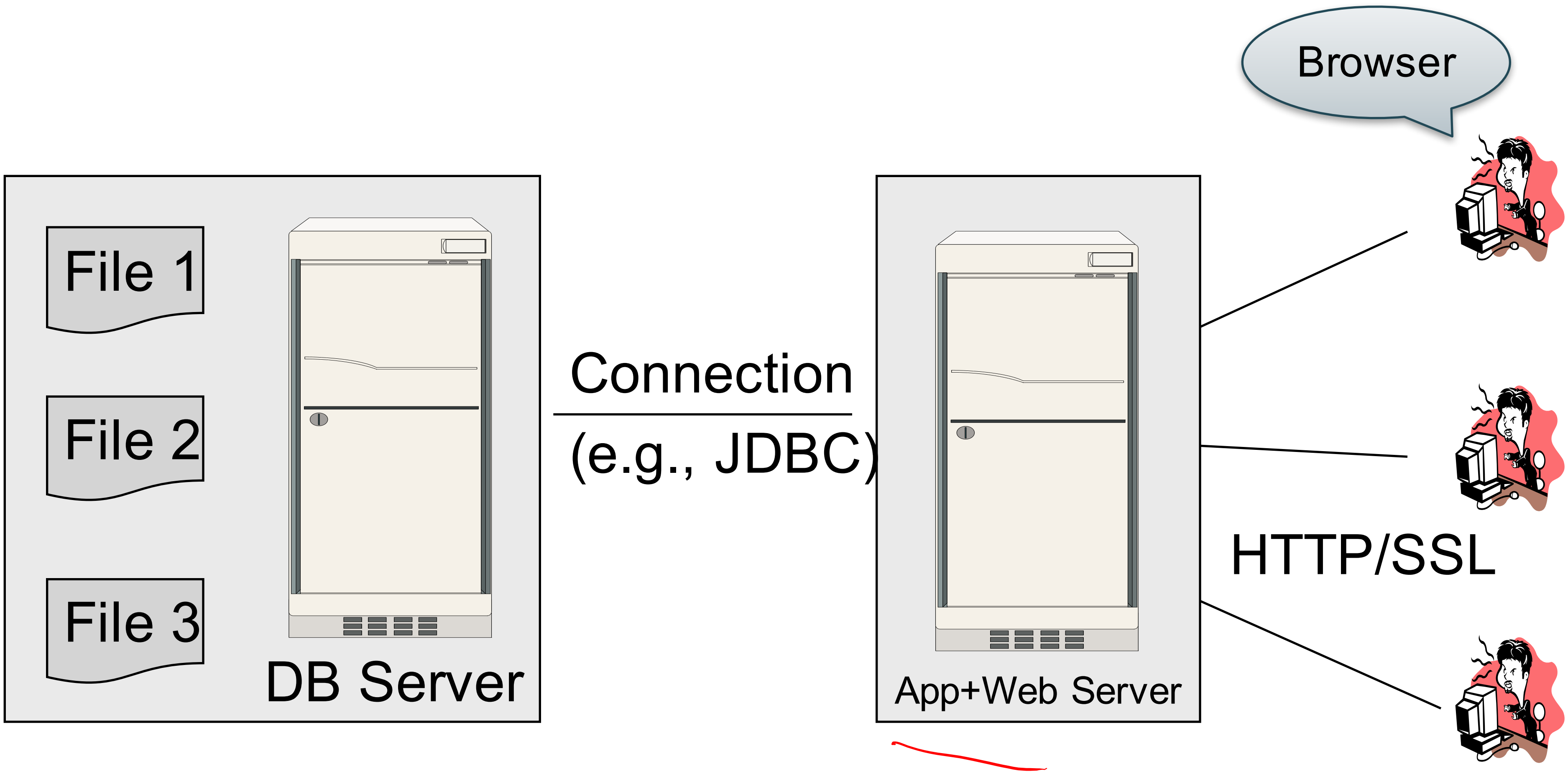
Client-Server

- 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

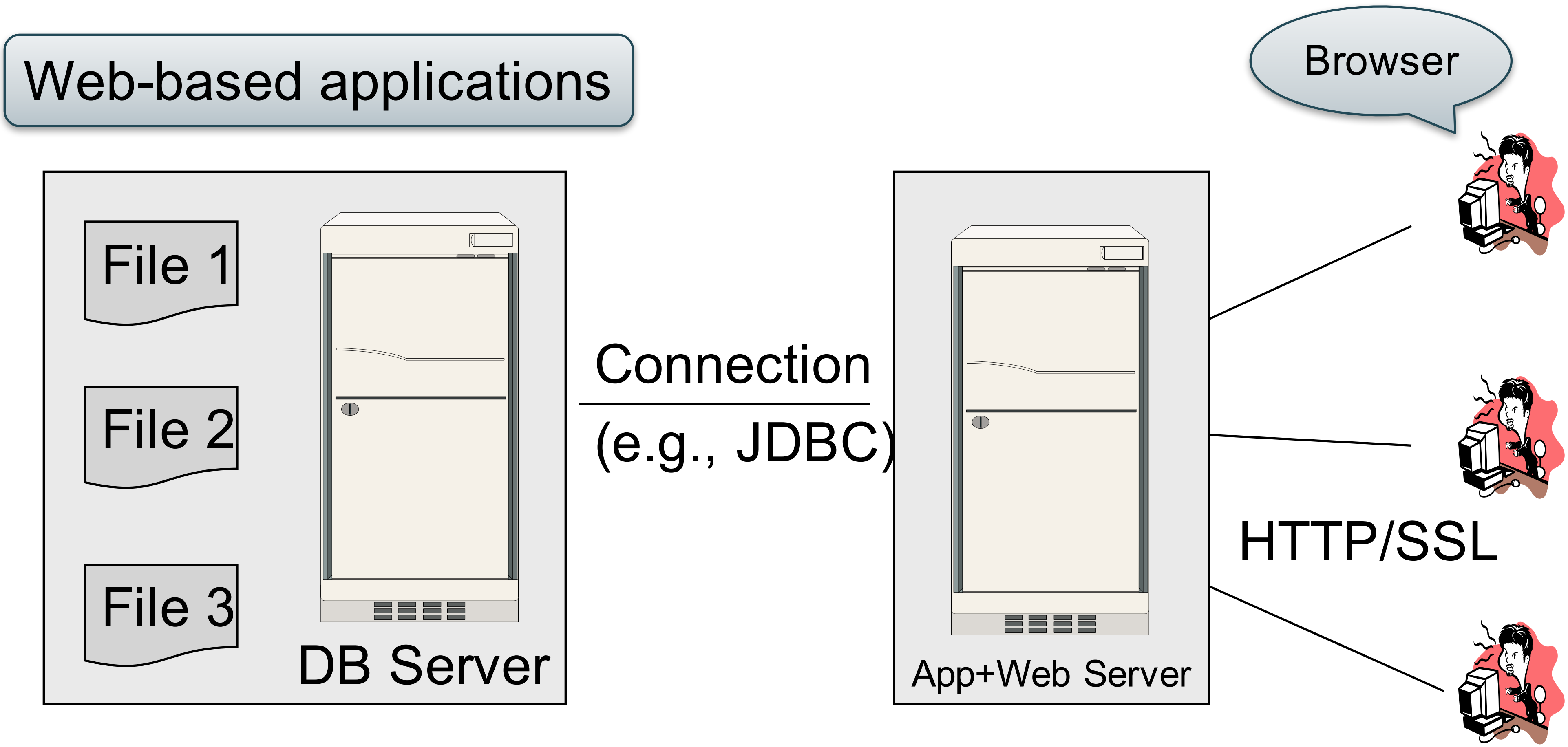
Web Apps: 3 Tier



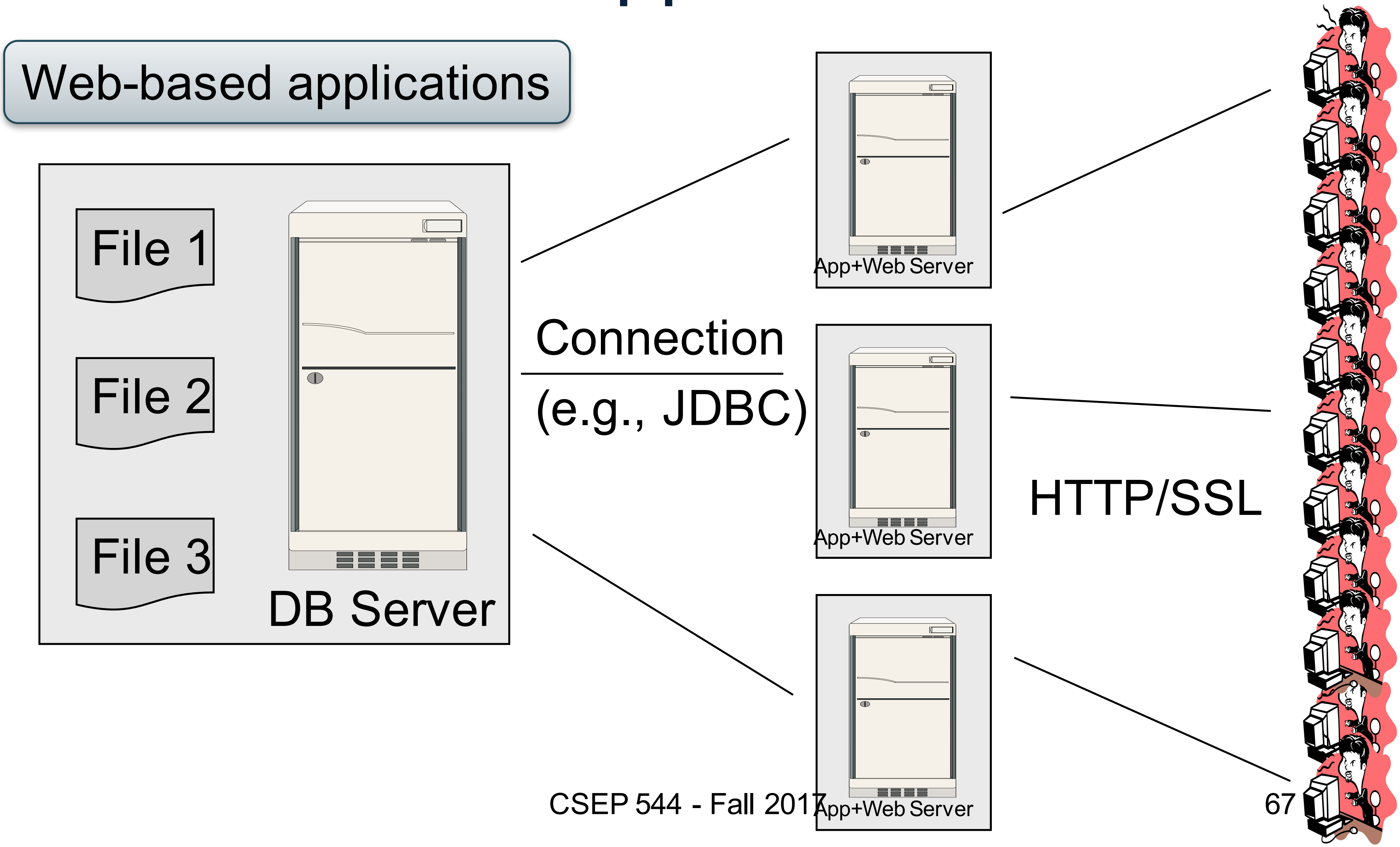
Web Apps: 3 Tier



Web Apps: 3 Tier



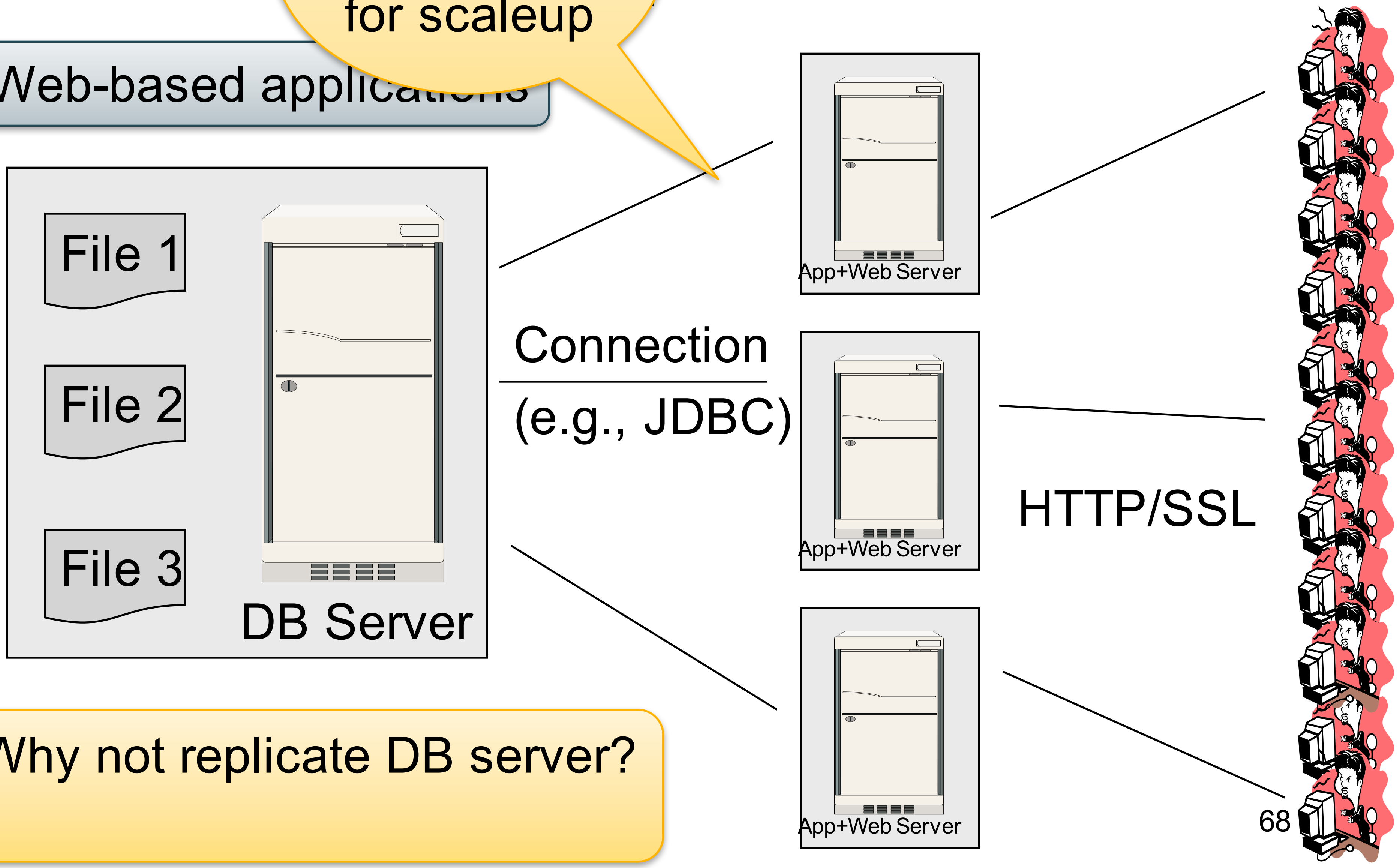
Web Apps: 3 Tier



...ps: 3 Tier

Replicate App server for scaleup

Web-based applica...

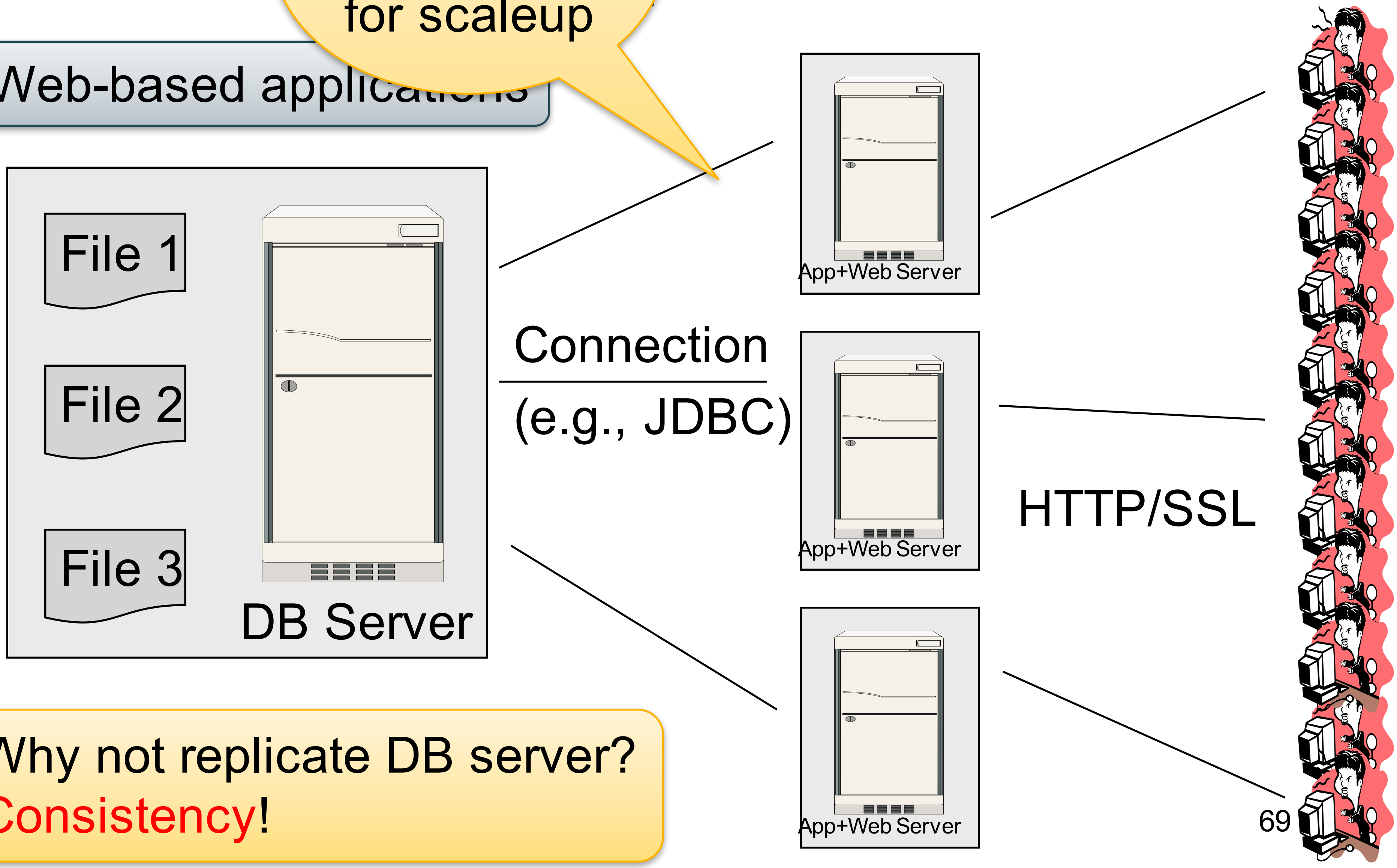


Why not replicate DB server?

ps: 3 Tier

Replicate App server for scaleup

Web-based applications



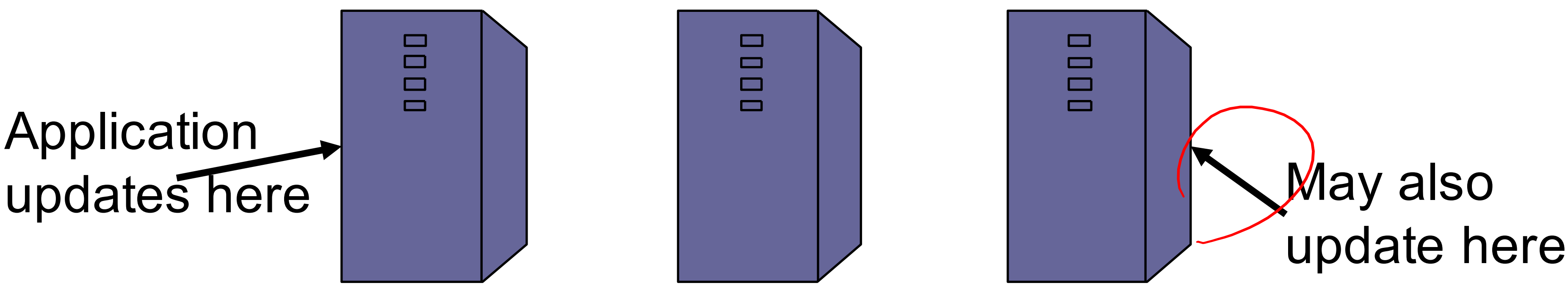
Why not replicate DB server?
Consistency!

Replicating the Database

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

Scale Through Partitioning

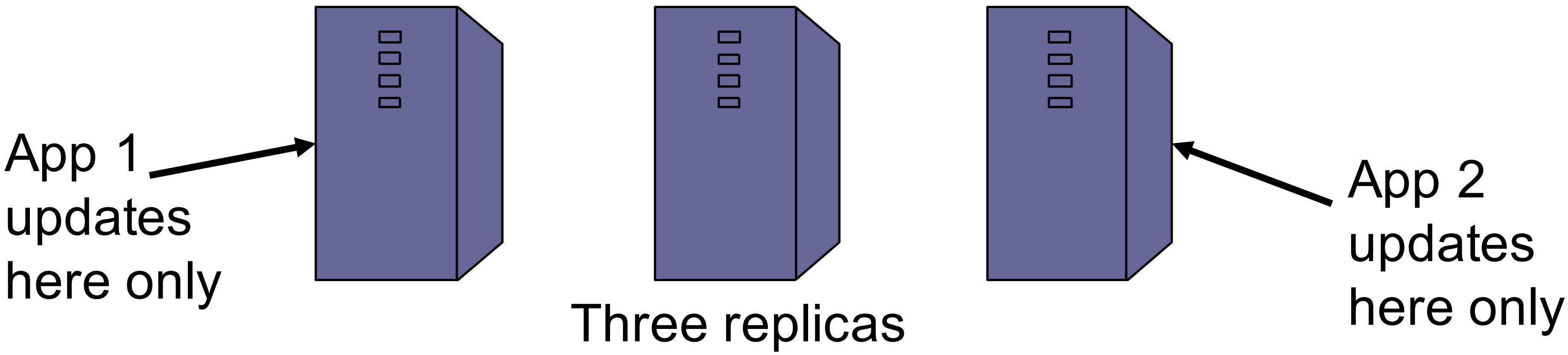
- Partition the database across many machines in a cluster
  - Database now fits in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for writes but reads become expensive!



Three partitions

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)
  - 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 (surprise!)
- 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

A Case Study: AsterixDB

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
  - 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 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"
}
```

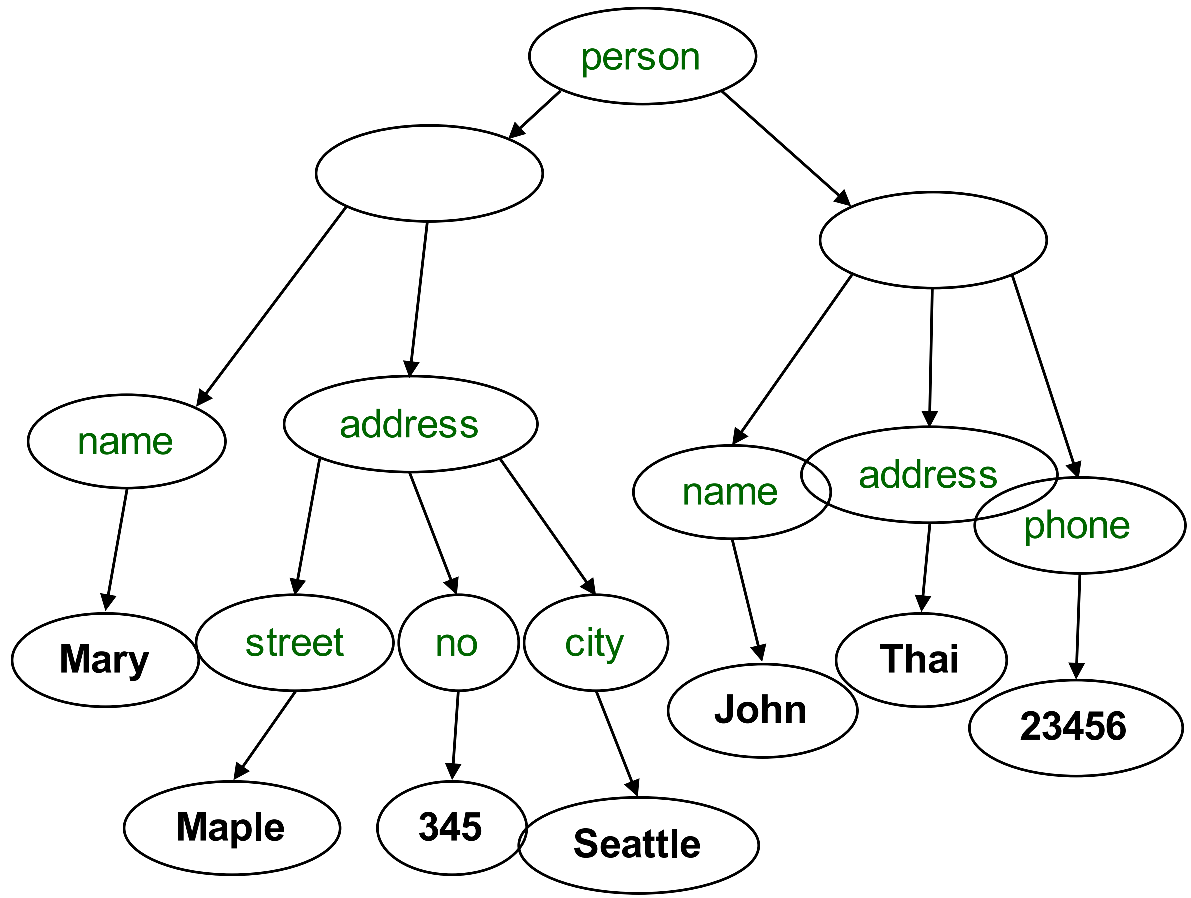
```
{"id":"07",
   "title": "Databases",
   "author": ["Garcia-Molina",
              "Ullman",
              "Widom"]
}
```

JSon Datatypes

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

JSon Semantics: a Tree !

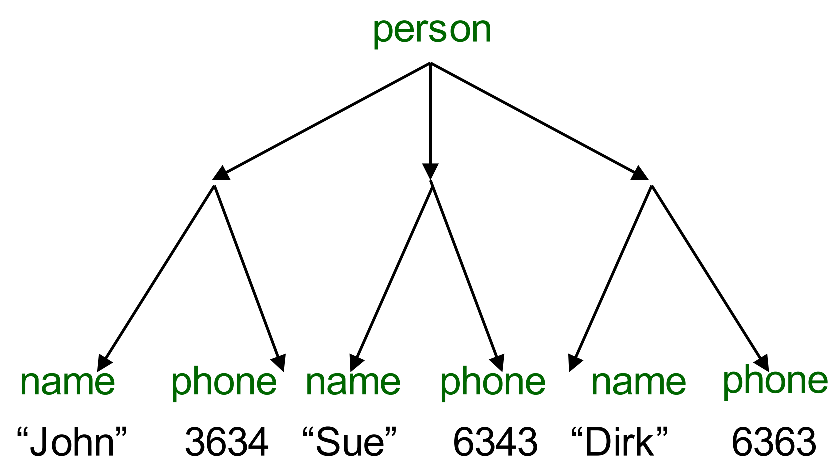
```
{"person":
  [{"name": "Mary",
     "address":
       {"street":"Maple",
         "no":345,
         "city": "Seattle"}},
    {"name": "John",
     "address": "Thailand",
     "phone":2345678}}
  ]
}
```



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

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 } }
{ }
```

ADM Language: SQL++

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

- DML: select-from-where

Dataverse

A Dataverse is a Database

```
CREATE DATAVERSE lecp544
CREATE DATAVERSE lecp544 IF NOT EXISTS

DROP DATAVERSE lecp544
DROP DATAVERSE lecp544 IF EXISTS

USE lecp544
```

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 lecp544;
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 lecp544;
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 lecp544;
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 lecp544;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   email: string?
}
```

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

```
USE lecp544;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

Dataset with Auto Generated Key

```
USE lecp544;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   myKey: uuid,
   Name : string,
   email: string?
}
```

```
USE lecp544;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;
```

{"Name": "Alice"}
{"Name": "Bob"}
…

Note: no myKey since it will be 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

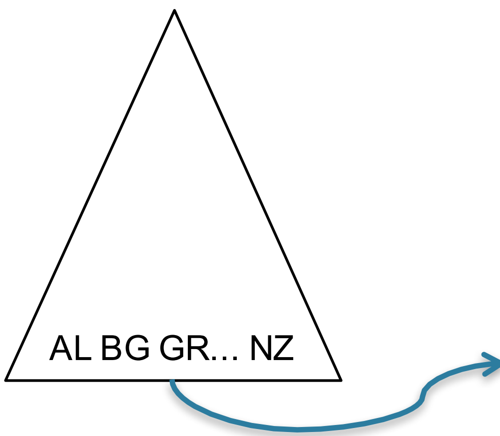
- Can declare an index on an attribute of a top-most collection
- Available:
  - BTREE: good for equality and range queries
    E.g. name="Greece"; 20 < age and age < 40
  - RTREE: good for 2-dimensional range queries
    E.g. 20 < x and x < 40 and 10 < y and y < 50
  - KEYWORD: good for substring search

Indexes

Cannot index inside a nested collection

```
USE lecp544;
CREATE INDEX countryID
    ON country(`-car_code`)
    TYPE BTREE;
```

```
USE lecp544;
CREATE INDEX cityname
    ON country(city.name)
    TYPE BTREE;
```



Country:

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

SQL++ Overview

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

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

world

Retrieve Everything

```
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, …*],

Retrieve countries, one by one

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

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

Answer

```
country1
country2
...
```

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

Escape characters

> "-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);
```

Just the arrays

The problem:

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

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);
```

The problem:

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

Just the objects

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