

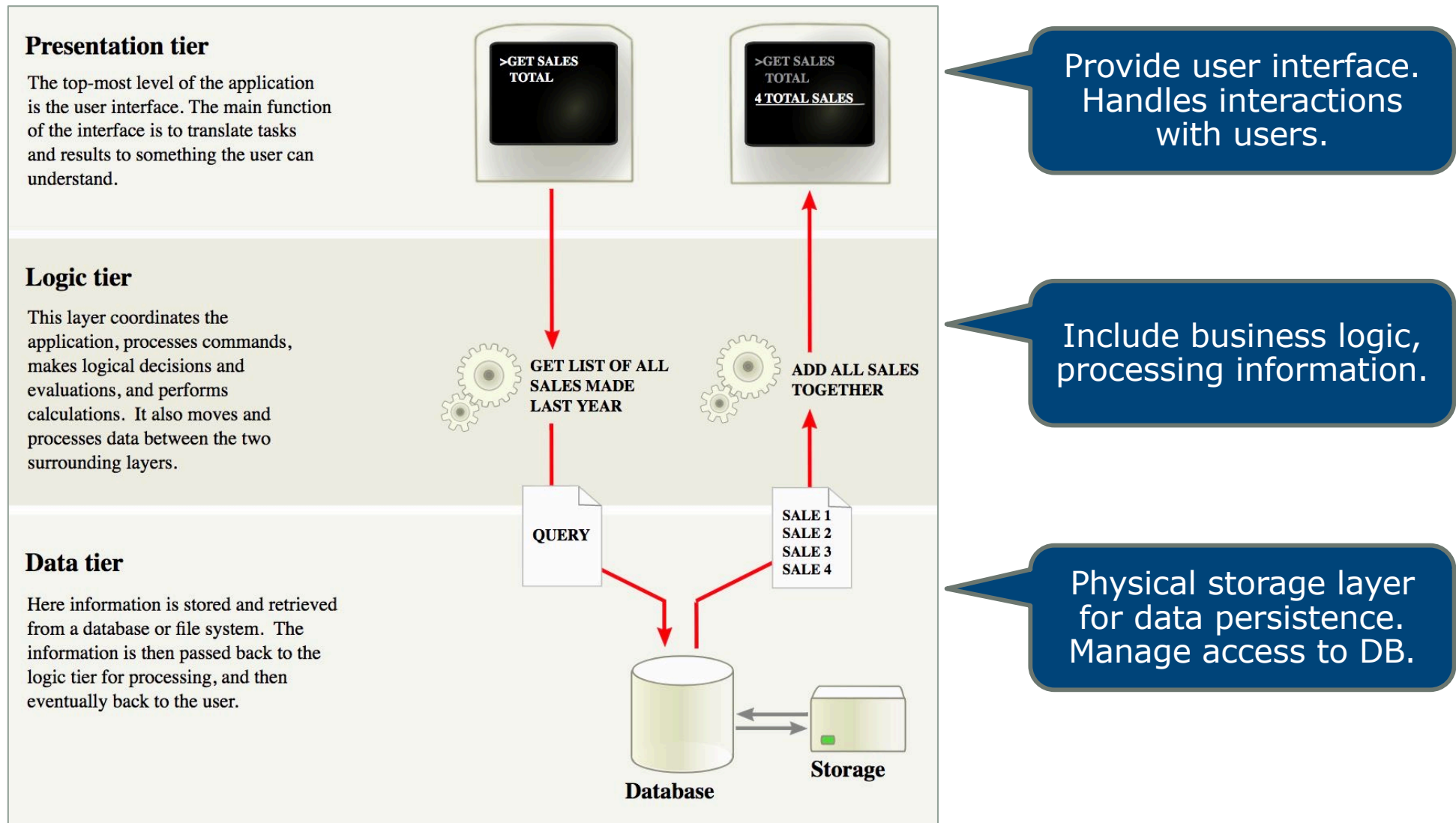
NoSQL Database

CS 4750 Database Systems

2 Main Types of Data Management

OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Transaction-heavy system	Retrieving and analysis system
Insert, update, delete info	Extract data for analyzing
Many simple lookup or single-join queries	Many joins and aggregations
Many small updates and inserts	Little to no updates
Managing consistency is crucial	Query optimization is crucial
Tables are (by default) normalized	Tables are (by default) not normalized

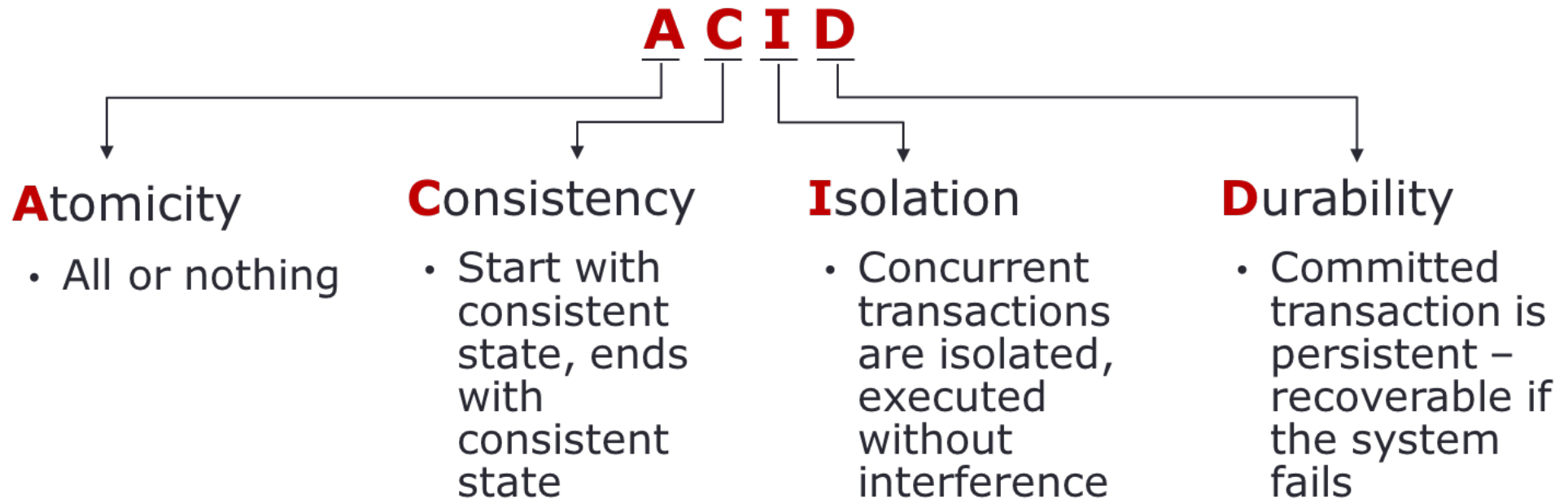
3-Tier Architecture for OLTP



[ref: https://en.wikipedia.org/wiki/Multitier_architecture]

RDBMS and ACID Properties

Four properties of transactions that a DBMS follows to handle concurrent access while maintaining consistency



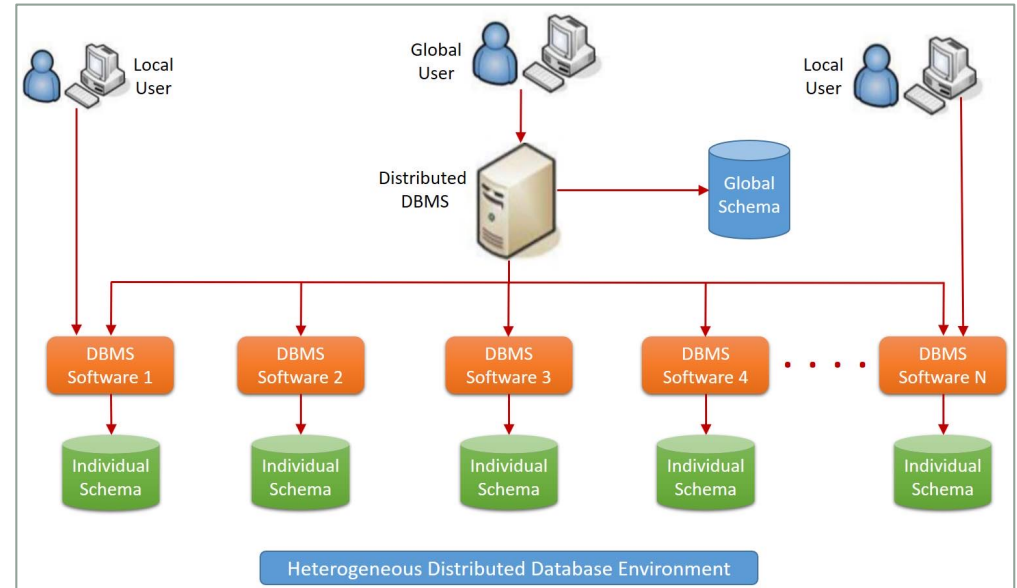
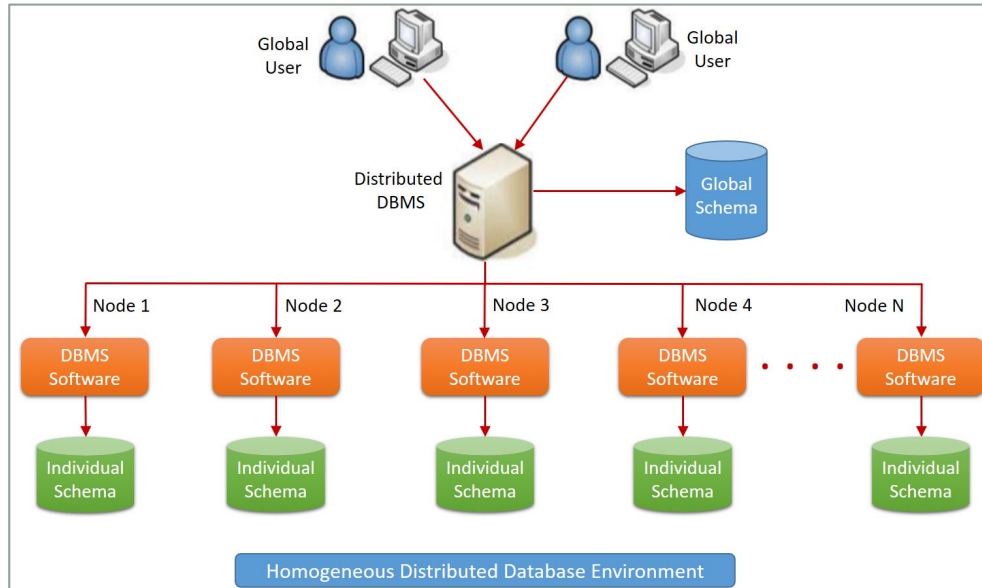
Challenges with RDBMS

- Low scalability
- Distributed DB
- Rigid schema / flat data
 - No arrays
 - No complex data type or objects
 - No missing / extra attributes

Scaling Issues in Centralized DB

- As a DB gets bigger, we try to scale a DB server until a DB become bottleneck.
- One way to solve the performance issues is to change from a centralized DB to distributed DBs.

Distributed DBs



- Fragmentation: need to coordinate operations across fragments
- Replication: need to synch to prevent inconsistent version
- Achieving ACID is challenging

ACID work in a centralized database system,
not in a distributed database system

[Ref: images from Pattamsetti, "Distributed Computing in Java 9"]

CAP Theorem

- **Consistency** -- All copies (across nodes) have the same value
- **Availability** -- System can still function even if some nodes fail
- **Partition tolerance** -- System can function even if communication between nodes (the partitions reside) fails
 - Network can break into two or more parts, each with active systems that communicate with the other parts
- Must have **exactly two** of the three properties for any system
- Very large system will partition by default, thus **choose one of consistency or availability**
 - Traditional database – choose **consistency**
 - Most web apps – choose **availability** (except some specific/important parts such as order/payment processing)

Threats on CAP

- Only two of the three properties are guarantees:
 - **Consistency** – every read receives the most recent write or an error
 - **Availability** – every request must respond with a non-error
 - **Partition tolerance** – continued operation in presence of dropped or delayed message
- **Distributed RDBMS** – partition tolerance + **consistency**

Intended to be highly consistent – but may sacrifice some consistency to boost availability

- **NoSQL systems** – partition tolerance + **availability**

Intended to be highly available – but may sacrifice some availability to boost consistency

Achieving CAP can be very difficult with the growth of data. Instead of using ACID or CAP, we may use a more relaxed set of properties, **BASE**

BASE Consistency Model

- With the enormous growth in data, achieving ACID or CAP becomes very difficult.
- A more relaxed set of properties is **BASE**
- **B**asically **A**vailable, **S**oft state, **E**ventually consistent

Most failures do not cause a complete system outage

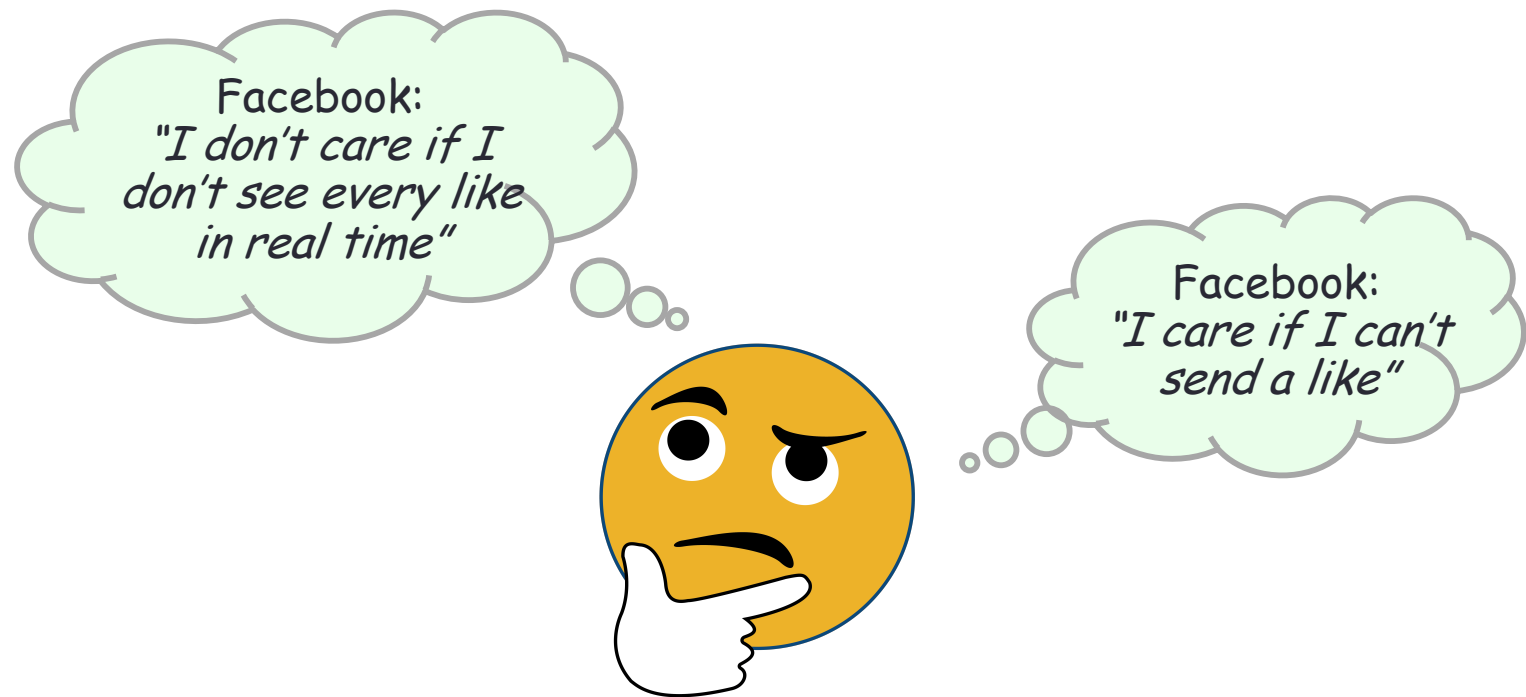
System is not always write-consistent

Data will eventually converge to agreed values

- Key idea:
 - Databases may not all be in the same state at the same time ("soft state")
 - After synchronization is complete, the state will be consistent

NoSQL – For Scaling and Flexibility

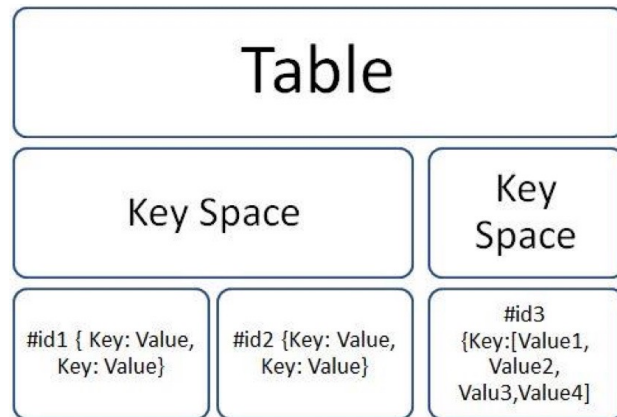
- Loose data model
- Give up built-in OLAP/analysis functionality
- Give up built-in ACID consistency
- Rely on **BASE** consistency model



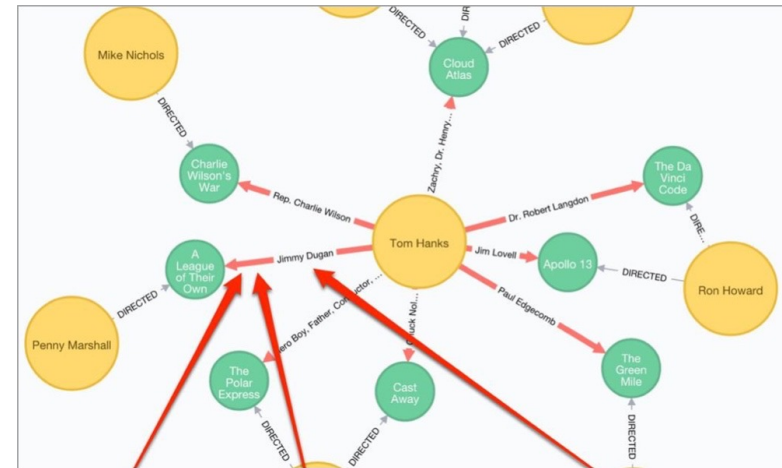
[Ref: emoji by Ekarin Apirakthanakorn]

NoSQL Data Models

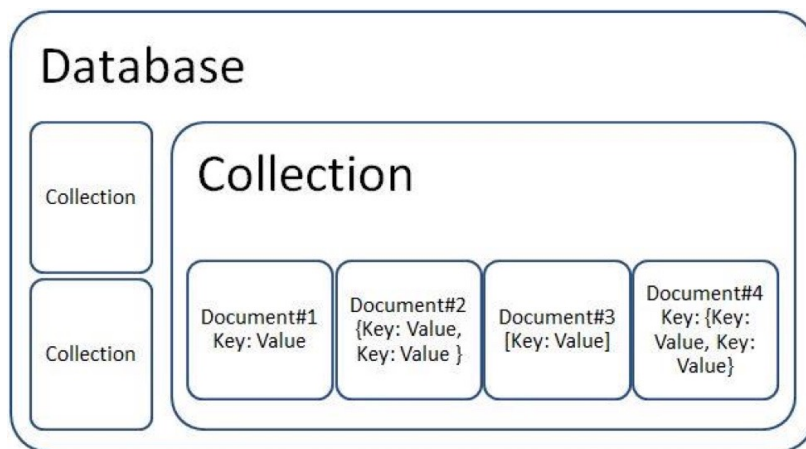
Key-value



Graph



Document



Column-family

Table			
Column Family 1		Column Family 2	Column Family 3
Column 1	Column 2	Column 3	Column 4
#1 {Key: Value, Key: Value }	#1 {Key: Value, Key: Value }		#1 {Key: Value, Key: Value }
#2 {Key: Value, Key: Value }	#2 {Key: Value, Key: Value }	#2 {Key: Value, Key: Value }	#2 {Key: Value, Key: Value }

Example NoSQL Data Models Implementation

Key-value



amazon
DynamoDB



redis



RocksDB

Graph



neo4j



**Amazon
Neptune**

Document



mongoDB

Column-family



**Google Cloud
Bigtable**



cassandra

NoSQL: Key-Value

- (key, value) pairs
- Key can be string, integer, ..., unique for the entire data set
- Value can be any type
- Basic operations:
 - `get(key)` – returns value
 - `put(key, value)` – add (key, value) pair to the data set
- Example flight information as key-value pairs

key	value
flightNumber	Complete record of a particular flight
date	All flight records on a particular date
(origin, destination, date)	All flight records between the origin and the destination on a particular date

NoSQL: Document

- Data set can be any kinds of files that are parsable
 - Structured document: CSV
 - Semi-structured document: XML, JSON
- Human-readable, may be unordered, heterogeneous data, fields may be skipped
- Example friend information as XML and JSON

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<friends>
  <friend>
    <name>Humpty</name>
    <email>humpty@uva.edu</email>
    <phone>434-111-1111</phone>
    <photo>images/friend1.png</photo>
  </friend>
  <friend>
    <name>Dumpty</name>
    <email>dumpty@uva.edu</email>
    <phone>434-222-2222</phone>
    <photo>images/friend2.png</photo>
  </friend>
</friends>
```

JSON

```
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "434-111-1111",
      "photo": "images/friend1.png"
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "434-222-2222",
      "photo": "images/friend2.png"
    }
  ]
}
```

Benefits of Document NoSQL

- Unopinionated
- Good for big data
- Faster reads / data retrieval (no joins)
- Extensible / flexible schema
- Data in developer-friendly format
 - Dictionary / JSON layout
 - Data is (most often) grouped by entity
 - Easy to parse with loops
 - More intuitive to understand
 - Generally models the real world a bit better

SQL vs. NoSQL

	SQL	NoSQL
Properties	<ul style="list-style-type: none">- ACID	<ul style="list-style-type: none">- BASE
Language	<ul style="list-style-type: none">- Standardized	<ul style="list-style-type: none">- Varies widely
Scaling	<ul style="list-style-type: none">- Vertical scaling	<ul style="list-style-type: none">- Horizontal scaling
Structure	<ul style="list-style-type: none">- Tabular	<ul style="list-style-type: none">- Hierarchical- Graph- Key-value
Schema	<ul style="list-style-type: none">- Highly opinionated- Very rigid	<ul style="list-style-type: none">- Little/no opinion- Very flexible
Support	<ul style="list-style-type: none">- Plenty/straightforward	<ul style="list-style-type: none">- Less than SQL (as it emerged later)

JSON (JavaScript Object Notation)

- **Data representation** for storing and exchanging between server and client
- Looks like JavaScript object, but it is just **plain text data** (not an object)
- **Light weight**
 - Plain text, containing only data to be transferred → fast & easy to load
- **Scalable**
 - Flexible, semi-structure, extensible
- **Standard structure**
 - Easy to distribute data over the Internet
- **Multiple applications**
 - JSON data resource can easily be reused to generate different view (promoting MVC)

```
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "111-111-1111",
      "age": 20
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "222-222-2222",
      "age": 21,
      "BOD": "11/16/2000"
    }
  ]
}
```

How does JSON work?

```
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "111-111-1111",
      "age": 20
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "222-222-2222",
      "age": 21,
      "BOD": "11/16/2000"
    }
  ]
}
```

- Data are presented in property **name-value pairs**
- Strings and property names (or keys) must be placed in **quotes**
- The key is separated from its value by a **colon**
- Each key-value pair is separated by a **comma**. No after the last key-value pair.

How does JSON work? (cont.)

```
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "111-111-1111",
      "age": 20
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "222-222-2222",
      "age": 21,
      "BOD": "11/16/2000"
    }
  ]
}
```

Values can be any of the following data types:

- **String** – text (must be in double quotes)
- **Number**
- **Boolean**
- **Array** – array of values or objects; enclosed by []
- **Object** – JavaScript object (can contain child objects or arrays); enclosed by {}
- **Null** – when the value is empty or missing

How does JSON work? *(cont.)*

```
{  
  "name": "Humpty",  
  "email": "humpty@uva.edu",  
  "phone": "111-111-1111",  
  "phone": "222-222-2222",  
  "age": 20  
}
```



Duplicate keys are not allowed

```
{  
  "name": "Humpty",  
  "email": "humpty@uva.edu",  
  "phone": [ "111-111-1111",  
             "222-222-2222" ],  
  "age": 20  
}
```

Use an array instead

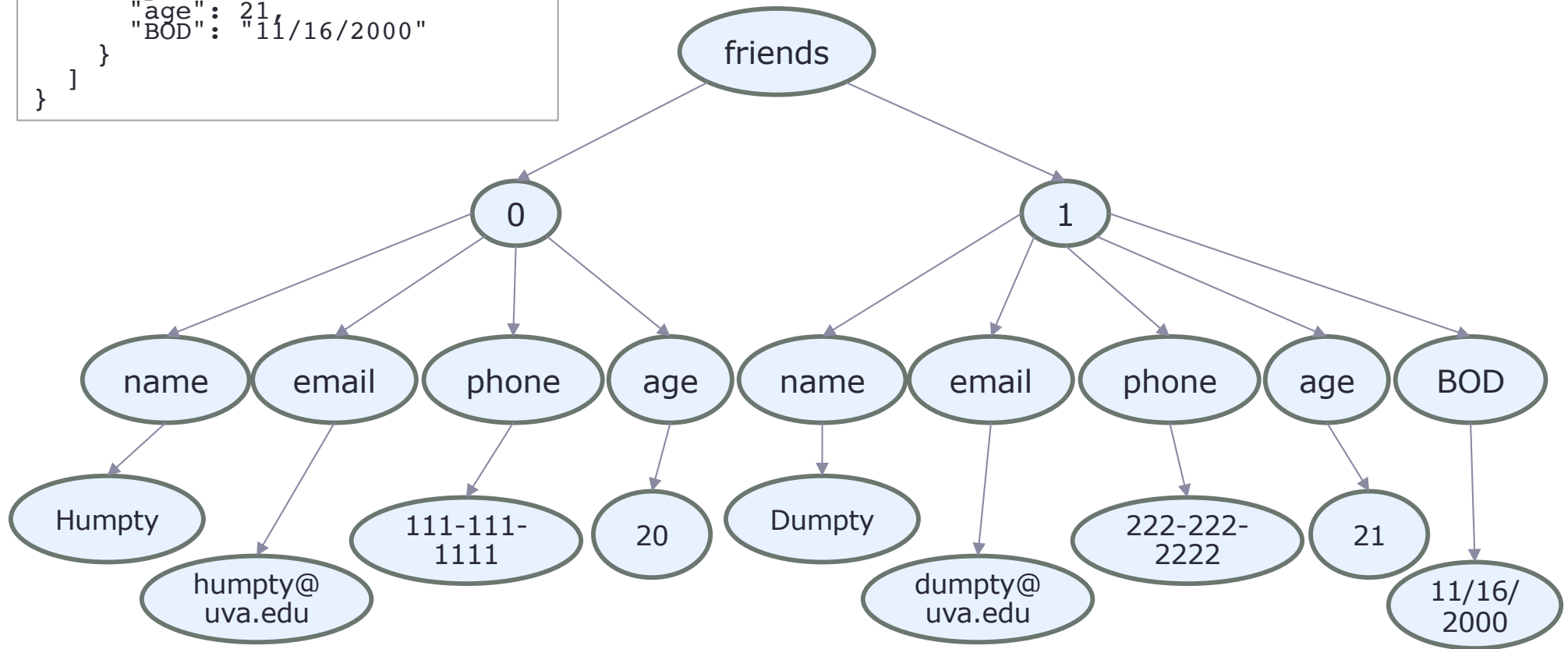
More JSON Example

```
{  
  "events": [  
    {  
      "place": "Charlottesville, VA",  
      "date": "July 20",  
      "map": "img/map-va.png"  
    },  
    {  
      "place": "Austin, TX",  
      "date": "July 23",  
      "map": "img/map-tx.png"  
    },  
    {  
      "place": "New York, NY",  
      "date": "July 30",  
      "map": "img/map-ny.png"  
    }  
  ]  
}
```

-  Object
-  Array

Semi-Structured Data → Tree

```
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "111-111-1111",
      "age": 20
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "222-222-2222",
      "age": 21,
      "BOD": "11/16/2000"
    }
  ]
}
```

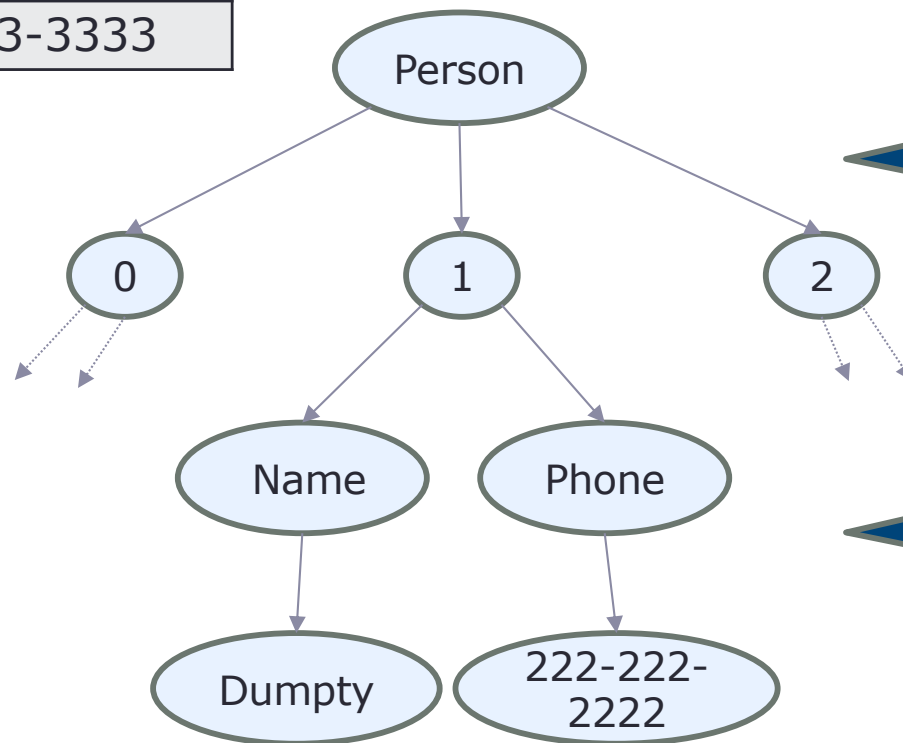


From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	333-333-3333

How is a table mapped to a semi-structured document?



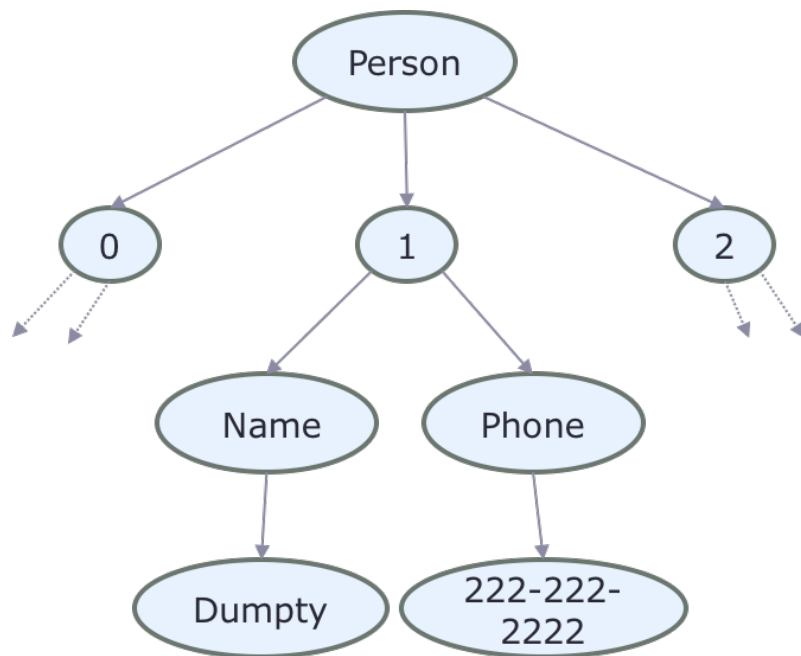
A table is just an array of elements (rows)

Rows are just simple (unnested) objects

From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	333-333-3333



```
{
  "Person": [
    {
      "Name": "Humpty",
      "Phone": "111-111-1111",
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
    },
    {
      "Name": "Wacky",
      "phone": "333-333-3333"
    }
  ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	NULL

How can NULL
be represented?

```
{
  "Person": [
    {
      "Name": "Humpty",
      "Phone": "111-111-1111",
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
    },
    {
      "Name": "Wacky",
      "phone": NULL
    }
  ]
}
```

From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	NULL

How can NULL
be represented?

```
{
  "Person": [
    {
      "Name": "Humpty",
      "Phone": "111-111-1111",
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
    },
    {
      "Name": "Wacky"
    }
  ]
}
```

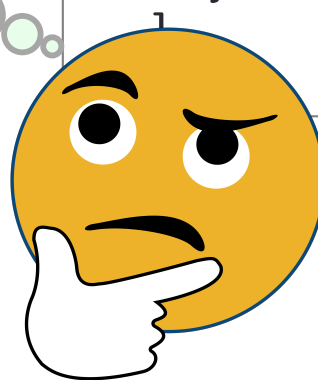
Ok if a field is
missing

From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	333-333-3333

*Are there things that
the relational model
cannot represent?*



```
{  
  "Person": [  
    {  
      "Name": "Humpty",  
      "Phone": "111-111-1111",  
    },  
    {  
      "Name": "Dumpty",  
      "phone": "222-222-2222",  
    },  
    {  
      "Name": "Wacky",  
      "phone": "333-333-3333"  
    }  
  ]  
}
```

[Ref: emoji by Ekarin Apirakthanakorn]

From Relational to Semi-Structured

Person

Name	Phone
Humpty	[111-111-1111 , 111-111-1234]
Dumpty	222-222-2222
Wacky	333-333-3333

Things that the
Relational model
cannot represent

Non-flat data

```
{
  "Person": [
    {
      "Name": "Humpty",
      "Phone": [
        "111-111-1111",
        "111-111-1234"
      ]
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
    },
    {
      "Name": "Wacky",
      "phone": "333-333-3333"
    }
  ]
}
```

Array data
(non-flat data)

From Relational to Semi-Structured

Person

Name		Phone
fname	lname	111-111-1111
Humpty	Fuzzy	
Dumpty		222-222-2222
Wacky		333-333-3333

Things that the
Relational model
cannot represent

Non-flat data

```
{
  "Person": [
    {
      "Name": {
        "fname": "Humpty",
        "lname": "Fuzzy"
      },
      "Phone": "111-111-1111"
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
    },
    {
      "Name": "Wacky",
      "phone": "333-333-3333"
    }
  ]
}
```

Multi-part data
(non-flat data)

From Relational to Semi-Structured

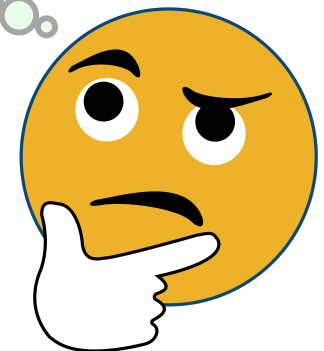
Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	333-333-3333

Orders

Name	Date	Dish
Humpty	11/11/2021	Taco
Dumpty	10/29/2021	Pizza
Dumpty	11/15/2021	Taco

*How do we represent
foreign keys?*



[Ref: emoji by Ekarin Apirakthanakorn]

From Relational to Semi-Structured

Person

Name	Phone
Humpty	111-111-1111
Dumpty	222-222-2222
Wacky	333-333-3333

Orders

Name	Date	Dish
Humpty	11/11/2021	Taco
Dumpty	10/29/2021	Pizza
Dumpty	11/15/2021	Taco

Nested foreign keys →
nested objects in JSON

Pros and Cons?

```
{
  "Person": [
    {
      "Name": "Humpty",
      "Phone": "111-111-1111",
      "Orders": [
        {
          "Date": "11/11/2021",
          "Dish": "Taco"
        }
      ]
    },
    {
      "Name": "Dumpty",
      "phone": "222-222-2222",
      "Orders": [
        {
          "Date": "10/29/2021",
          "Dish": "Piazza"
        },
        {
          "Date": "11/15/2021",
          "Dish": "Taco"
        }
      ]
    },
    {
      "Name": "Wacky",
      "phone": "333-333-3333"
    }
  ]
}
```

Wrap-Up

- E-R & relational data model – start with a schema.
- The data in a relational DB must fit the schema; the schema is known to the query processor.
- Traditional RDBMS uses SQL syntax and queries to retrieve and manipulate data.
- Sometimes, data are fuzzy and come in a semi-structured or unstructured format.
- Focusing on flexibility, we need loose data model (schemaless or semi-structured data model)
- The schemaless or semi-structured data model may make query processing harder.
- Relational data model – more well-defined; fixed schema; more efficient encoding
- NoSQL – less well-defined; flexible schema; easy data exchange

Wrap-Up (2)

- RDBMS – intended to be highly consistent (boost availability by sacrificing some consistency)
- NoSQL – intended to be highly available (boost consistency by sacrificing some availability)
- Relational database systems – **ACID**
- Distributed database systems – **CAP**
- NoSQL systems – **BASE**
- Most applications compromise, depending business logic
 - Consistency / availability
 - Scalability
 - Usability
 - Analysis requirements

No silver-bullet !!