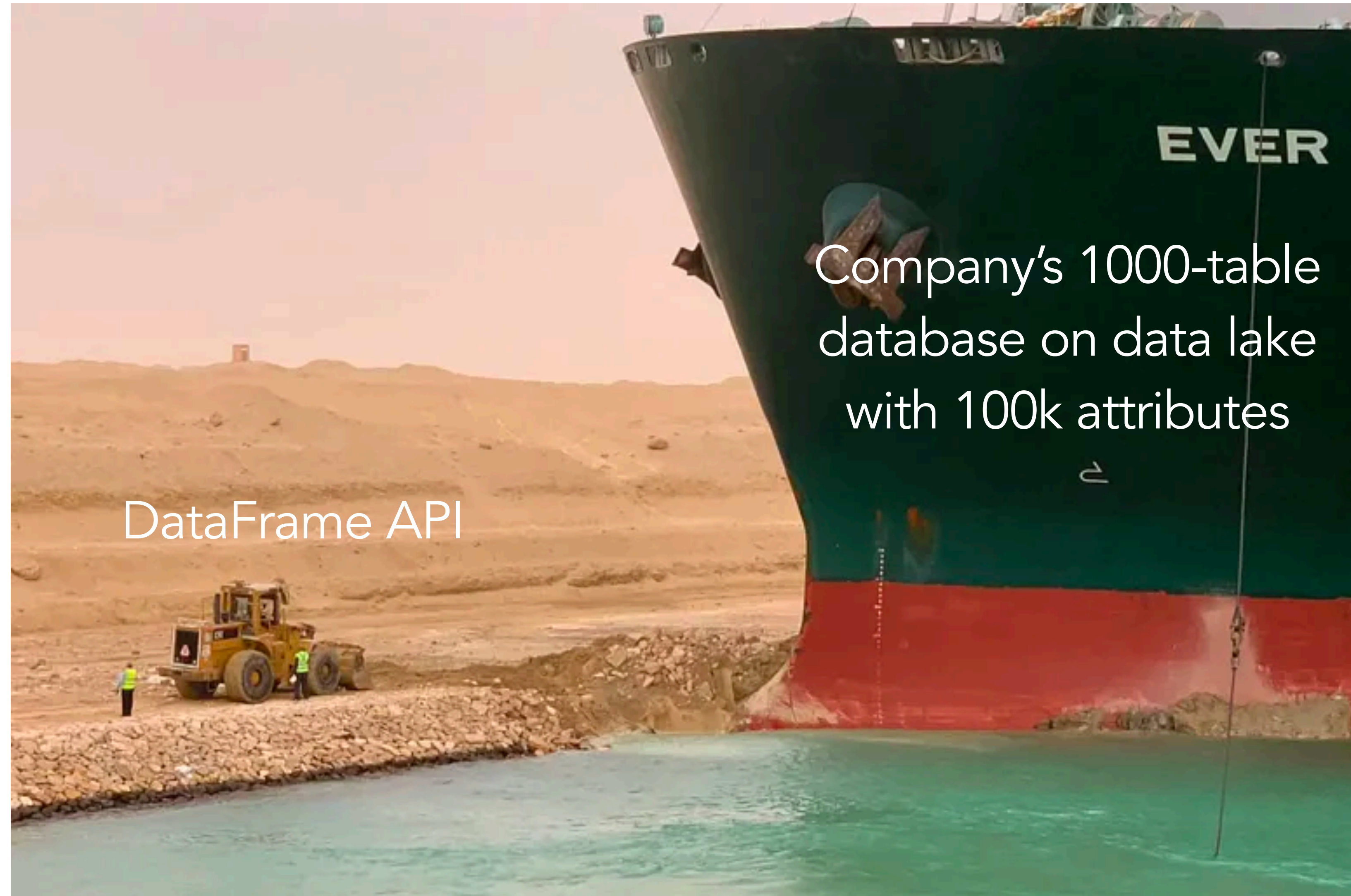


# DSC 204a Scalable Data Systems

- Haojian Jin



Company's 1000-table  
database on data lake  
with 100k attributes

DataFrame API

# Where are we in the class?

## Foundations of Data Systems (2 weeks)

- Digital representation of Data → Computer Organization → Memory hierarchy → Process → Storage

## Scaling Distributed Systems (3 weeks)

- Cloud → Network → Distributed storage → Parallelism → Partition and replication

## **Data Processing and Programming model (5 weeks)**

- **Data Models evolution** → Data encoding evolution → IO & Unix Pipes → Batch processing (MapReduce) → Stream processing (Spark)

# Today's topic: Data Model Evolutions & Query language

- Overview
- PIA (Distributed systems review)

# PA2

- Overview
  - PA.
    - PA0: Memory hierarchy (Dask).
    - PA1: Parallelism (Dask).
    - **PA2: Data processing (Spark).**
  - Lectures:
    - Foundations of DataSys (Data, Network, OS)
    - Scaling Distributed systems
    - Programming Interfaces

# Plan in the next few weeks

- Data models & query language
  - relational model
  - document model
  - graph model
- Encoding & Evolution
  - Json, XML, Protobuf
- Data Flow
  - REST and RPC
  - Message-passing data flow

# Plan in the next few weeks

- Batch processing
  - Unix pipes
  - MapReduce
- Stream processing
  - Spark
- Emerging topics

# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- Many-to-many v.s. one-to-many
- Hierarchical/network/relational model
- Query languages

# Overview

- Data models: not only on how the software is written, but also on how we think about the problem that we are solving.
- Most applications are built by layering one data model on top of another.
  - For each layer, the key question is: how is it represented in terms of the next-lower layer? each layer hides the complexity of the layers below it by providing a clean data model.
- General-purpose data models for data storage and querying
  - Relational DB: IBM DB2, MS SQL Server, PostgreSQL
  - Document DB: MongoDB, RethinkDB, CouchDB, Espresso
  - Graph-based DB: Neo4j, Titan, InfiniteGraph



# Relational Model

- Data is organized into relations (called tables in SQL), where each relation is an unordered collection of tuples (rows in SQL).
- The goal of the relational model was to hide that implementation detail behind a cleaner interface.
- Optimization opportunities => Better performance!

# The Birth of NoSQL (Not Only SQL | Document Model)

- A need for **greater scalability** than relational databases can easily achieve, including very large datasets or very high write throughput
- A widespread preference for **free and open source** software over commercial database products
- **Specialized query operations** that are not well supported by the relational model
- Frustration with the restrictiveness of relational schemas, and a desire for a more **dynamic and expressive** data model

# Relational Model Versus Document Model

**Polyglot persistence:** It therefore seems likely that in the foreseeable future, relational databases will continue to be used alongside a broad variety of nonrelational datastores

# The Object-relational Mismatch

- If data is stored in relational tables, **an awkward translation layer** is required between the objects in the application code and the database model of tables, rows, and columns.
- The disconnect between the models is sometimes called an **impedance mismatch**.

http://www.linkedin.com/in/williamhgates



## Bill Gates

Greater Seattle Area | Philanthropy

### Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

### Experience

Co-chair • Bill & Melinda Gates Foundation  
2000 – Present

Co-founder, Chairman • Microsoft  
1975 – Present

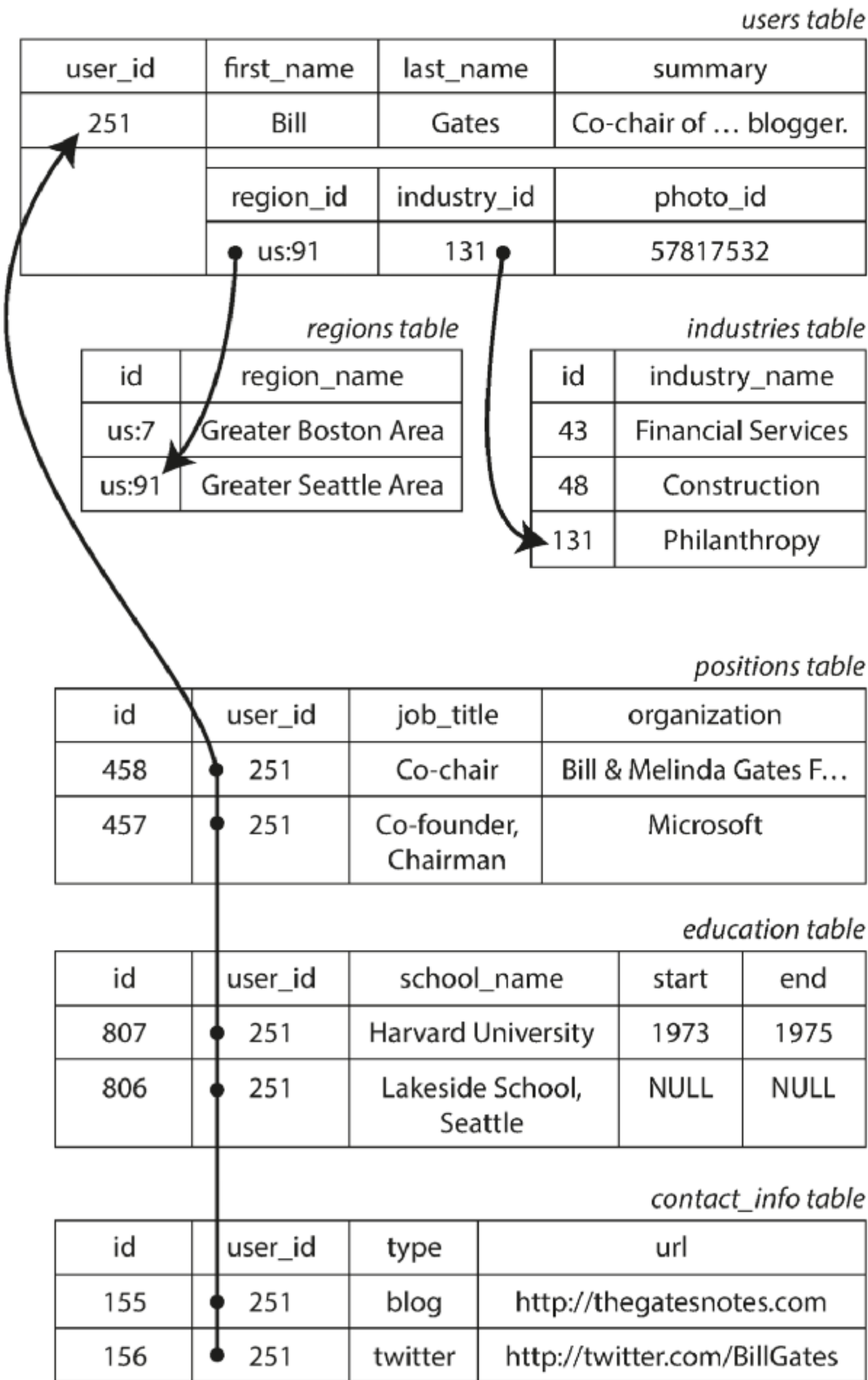
### Education

Harvard University  
1973 – 1975

Lakeside School, Seattle

### Contact Info

Blog: thegatesnotes.com  
Twitter: @BillGates



# The Object-relational Mismatch

- Traditional SQL model (prior to SQL:1999): the most common normalized representation is to put positions, education, and contact information in separate tables, with a **foreign key reference** to the users table.
- Later versions of the SQL standard added support for **structured datatypes and XML data**; this allowed multi-valued data to be stored within a single row, with support for querying and indexing inside those documents.
- A third option is to encode jobs, education, and contact info as a **JSON or XML document**, store it on a text column in the database, and let the application interpret its structure and content.



```

{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}

```

# Locality

- The JSON representation has better locality than the multi-table schema.
- If you want to fetch a profile in the relational example, you need to either perform multiple queries or perform a messy multiway join between the users table and its subordinate tables.
- In the JSON representation, all the relevant information is in one place, and one query is sufficient.



# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- Many-to-many v.s. one-to-many
- Hierarchical/network/relational model
- Query languages

# plain-text vs standardized lists for region/industrial

```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/255/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

# Standardized lists for region/industrial vs plain-text

- + Consistent style and spelling across profiles
- + Avoiding ambiguity (e.g., if there are several cities with the same name)
- + Ease of updating — the name is stored in only one place, so it is easy to update across the board if it ever needs to be changed (e.g., change of a city name due to political events)
- + Localization support — when the site is translated into other languages, the standardized lists can be localized, so the region and industry can be displayed in the viewer's language
- + Better search — e.g., a search for philanthropists in the state of Washington can match this profile, because the list of regions can encode the fact that Seattle is in Washington (which is not apparent from the string "Greater Seattle Area")

# Standardized lists for region/industrial vs plain-text

- + When you store the text directly, you are duplicating the human-meaningful information in every record that uses it.
  - + The advantage of using an ID is that because it has no meaning to humans, it never needs to change
  - + Removing such duplication is the key idea behind **normalization** in databases.
- 
- More complex data model
  - Hard to debug
    - A small project?
    - Normalization v.s. Denormalization?

# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- **Many-to-many v.s. one-to-many**
- Hierarchical/network/relational model
- Query languages

# Many-to-one and Many-to-Many Relationships



## Experience

### Co-chair

Bill & Melinda Gates Foundation

2000 – Present (13 years)

### Co-founder, Chairman



Microsoft

1975 – Present



### Education

Harvard University

1973 – 1975

## Microsoft

Come as you are. Do what you love. At Microsoft we help people and businesses throughout the world realize their full potential. We make this simple mission come to life every day through our ... [More »](#)

Co. Size: 10,001+ employees

Website: <http://www.microsoft.com/>

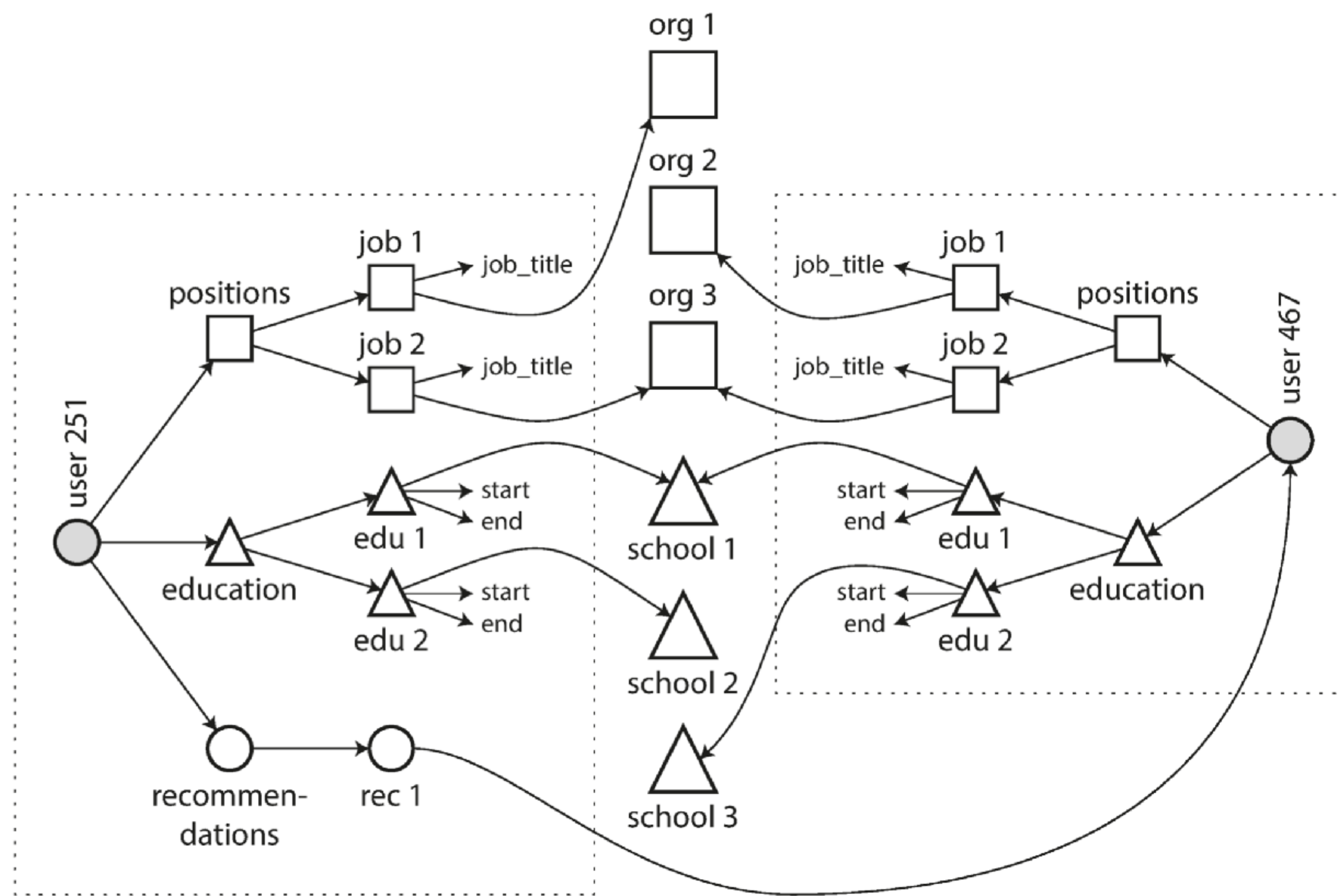
HQ: Greater Seattle Area

Industry: Computer Software

[Follow company](#) | [Careers](#)







The data within each dotted rectangle can be grouped into one document, but the references to organizations, schools, and other users need to be represented as references, and require joins when queried.

# Many-to-one and Many-to-Many Relationships

- Normalizing this data requires many-to-one relationships, which don't fit nicely into the document model.
- Organizations and schools as entities: In the previous description, organization (the company where the user worked) and school\_name (where they studied) are just strings. Perhaps they should be references to entities instead?
- Recommendations: one user can write a recommendation for another user. The recommendation is shown on the résumé of the user who was recommended, together with the name and photo of the user making the recommendation. If the recommender updates their photo, any recommendations they have written need to reflect the new photo.



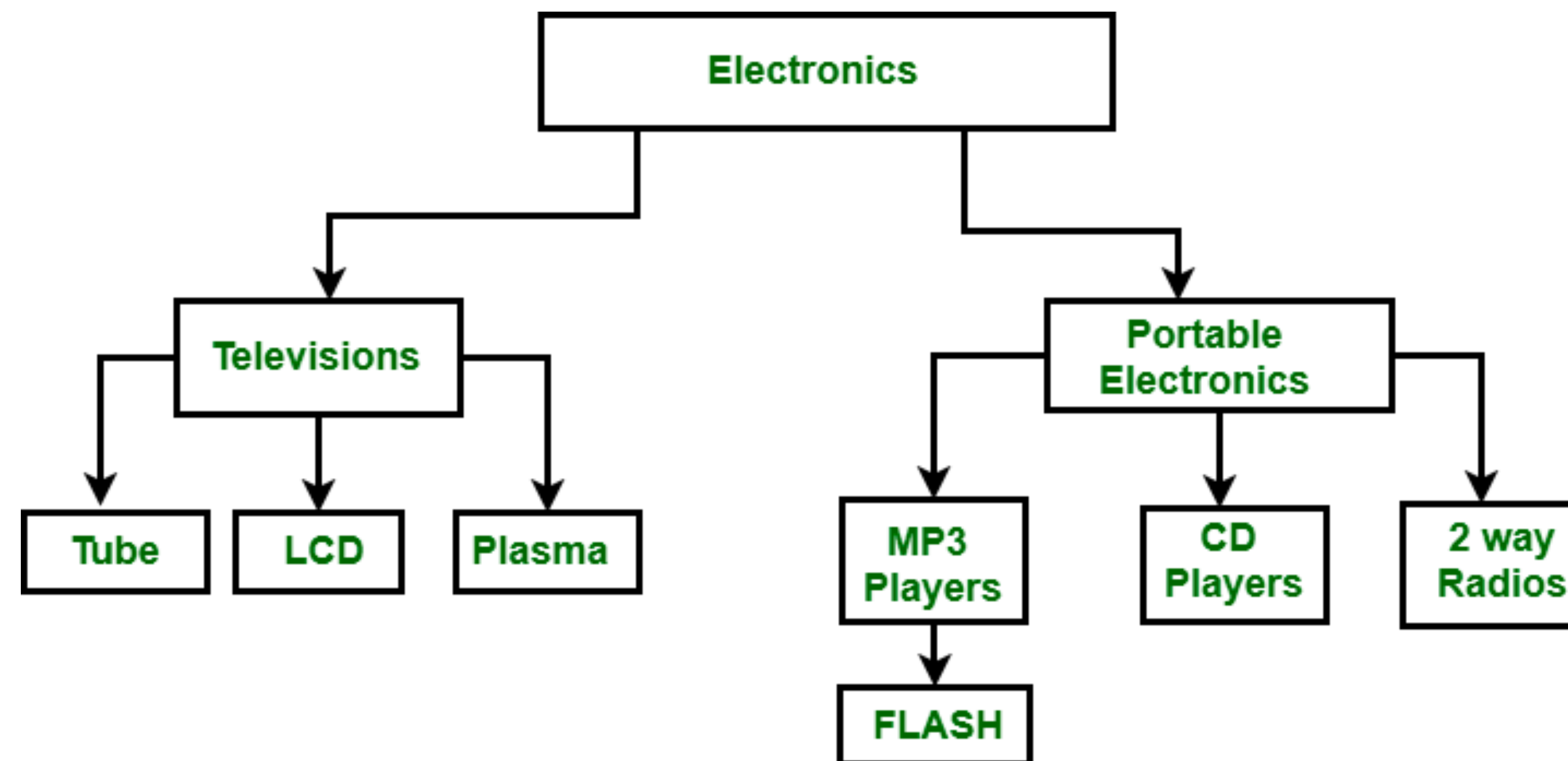
# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- Many-to-many v.s. one-to-many
- Hierarchical/network/relational data model
- Declarative | imperative query languages

# Are Document Databases Repeating History?

While **many-to-many relationships** and **joins** are routinely used in relational databases, document databases and NoSQL reopened the debate on how best to represent such relationships in a database.

# Hierarchical data model



- every record has exactly one parent
- Worked well for one-to-many relationships
- Hard for many-to-many relationships
- Not support join.

# Network Model

- A record could have multiple parents.
- The links between records in the network model were not foreign keys, but more like pointers in a programming language.
- The only way of accessing a record was to follow a path from a root record along these chains of links. This was called an **access path**.
- Good for limited hardware capabilities in the 1970.
- Querying and updating are complicated and inflexible.
  - A side effect to too much freedom.

# The Relational Model

- Lay out all the data in the open: a relation (table) is simply a collection of tuples (rows).
- The query optimizer automatically decides which parts of the query to execute in which order, and which indexes to use.
- If you want to query your data in new ways, you can just declare a new index, and queries will automatically use whichever indexes are most appropriate.
- The relational model thus made it much easier to add new features to applications.
- You only need to build a query optimizer once, and then all applications that use the database can benefit from it.

# Relationships

- **One to many: e.g., Positions, education, contact\_info.**
  - Document databases ~ the hierarchical model: storing nested records.
- **Many to many: e.g., Organizations**
- **Many to one: e.g., five students are finishing one project.**
  - The related item is referenced by a unique identifier, which is called **a foreign key** in the relational model and **a document reference** in the document model. That identifier is resolved at read time by using **a join** or **follow-up queries**.

# Which data model leads to simpler application code?

- If the data in your application has a document-like structure -> good idea to use a document model.
  - A tree of one-to-many relationships.
  - Entire tree needs to read at once.
  - Partitioning (shredding) has a cost => can lead to cumbersome schemas.
- Limitation:
  - Cannot refer directly to a nested item (However, as long as documents are not too deeply nested, that is not usually a problem.)
  - The poor support for joins in document databases (depending on the application)
    - many-to-many relationships

# Schema flexibility in the document model

Most document databases, and the JSON support in relational databases, do not enforce any schema on the data in documents.

Schema-on-read (Document) is similar to dynamic (runtime) type checking in programming languages.

```
if (user && user.name && !user.first_name) {  
    // Documents written before Dec 8, 2013 don't have first_name  
    user.first_name = user.name.split(" ")[0];  
}
```



# Schema flexibility in the document model

Schema-on-write (Relational) is similar to static (compile-time) type checking.

Schema changes have a bad reputation of being slow and requiring downtime. => Not that bad.

Set to its default of NULL and fill it in at read time, like it would with a document database.

```
ALTER TABLE users ADD COLUMN first_name text;  
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL  
UPDATE users SET first_name = substring_index(name, ' ', 1); -- MySQL
```

# Data locality for queries

- If your application often needs to access the entire document, there is a performance advantage to this **storage locality**. The idea of grouping related data together for locality is not limited to the document model.

# Convergence of document and relational databases

- It seems that relational and document databases are becoming more similar over time, and that is a good thing: the data models complement each other.
- A hybrid of the relational and document models is a good route for databases to take in the future.

# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- Many-to-many v.s. one-to-many
- Hierarchical/network/relational data model
- **Declarative | imperative query languages**

# Query Languages for Data

- An **imperative** language tells the computer to perform certain operations in a certain order.
- In a **declarative** query language, like SQL or relational algebra, you just specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed—but not **how** to achieve that goal.
- SQL is a **declarative** query language, whereas IMS and CODASYL queried the database using **imperative** code.

# Query Languages for Data

```
function getSharks() {  
  var sharks = [];  
  for (var i = 0; i < animals.length; i++) {  
    if (animals[i].family === "Sharks") {  
      sharks.push(animals[i]);  
    }  
  }  
  return sharks;  
}
```

- declarative

```
SELECT * FROM animals WHERE family = 'Sharks';
```

- imperative

# More examples

```
<ul>
  <li class="selected"> ❶
    <p>Sharks</p> ❷
    <ul>
      <li>Great White Shark</li>
      <li>Tiger Shark</li>
      <li>Hammerhead Shark</li>
    </ul>
  </li>
  <li>
    <p>Whales</p>
    <ul>
      <li>Blue Whale</li>
      <li>Humpback Whale</li>
      <li>Fin Whale</li>
    </ul>
  </li>
</ul>
```

---

```
li.selected > p {
  background-color: blue;
}
```

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
  if (liElements[i].className === "selected") {
    var children = liElements[i].childNodes;
    for (var j = 0; j < children.length; j++) {
      var child = children[j];
      if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
        child.setAttribute("style", "background-color: blue");
      }
    }
  }
}
```

# Tradeoffs

- Declarative
  - More concise, easier to maintain
  - Performance optimization, parallel execution
  - Simple APIs, Hide low-level details
  - Hard to debug, limited degree of freedom
- Imperative
  - Easier to debug
  - More degree of freedom
  - Require more expertises



# Future directions: Codeless

The screenshot displays a codeless API development environment. At the top, there are tabs for 'ERS', 'CODING', and 'CODELESS'. A 'PUBLISH TO MARKETPLACE' button is visible in the top right corner. The main interface is divided into two sections: a logic editor on the left and a response preview on the right.

The logic editor is titled 'POST /findObjectsWithPaging'. It contains a sequence of steps for an API service named 'CodelessRecipes' and a method named 'findObjectsWithPaging':

- Set `resultList` to `create empty list`.
- Set `numberOfObjects` to `Get Object Count`, with `table name` set to `Method Argument "TableName"` and `where clause` set to `Method Argument "whereClause"`.
- Set `pagesQuantity` to `round up` of `numberOfObjects` divided by `100`.
- Set `offSet` to `0`.
- Set `sortBy` to `"position"`.
- Count with `i` from `0` to `pagesQuantity` by `1`.
- Do loop: Set `tableList` to `Load Table objects`, with `table name` set to `Method Argument "TableName"`, `where clause` set to `Method Argument "whereClause"`, `relations` set to `sortBy`, `relations depth` set to `100`, `page size` set to `100`, and `page offset` set to `offSet`.
- Set `offSet` to `offSet + 100`.
- For each item `i` in list `tableList`, do: in list `resultList` insert at `last` as `i`.
- Return `resultList`.

The right panel shows the 'RESPONSE' tab with a 'curl' environment. The curl command is:

```
curl -X "POST" "https://apitest.backendless.com/" \
-H "Content-Type: application/json" \
-H "Accept: application/json"
```

# Future directions: ChatGPT



Generate an sql query to exact the sum of sales in a standard employee table



Sure! To extract the sum of sales from a standard employee table, you would need to know the specific structure of your table, including the column names. However, I can provide you with a generic example based on common column names. Let's assume you have an "employees" table with columns such as "employee\_id" and "sales\_amount". You can use the following SQL query to calculate the sum of sales:



sql

Copy code

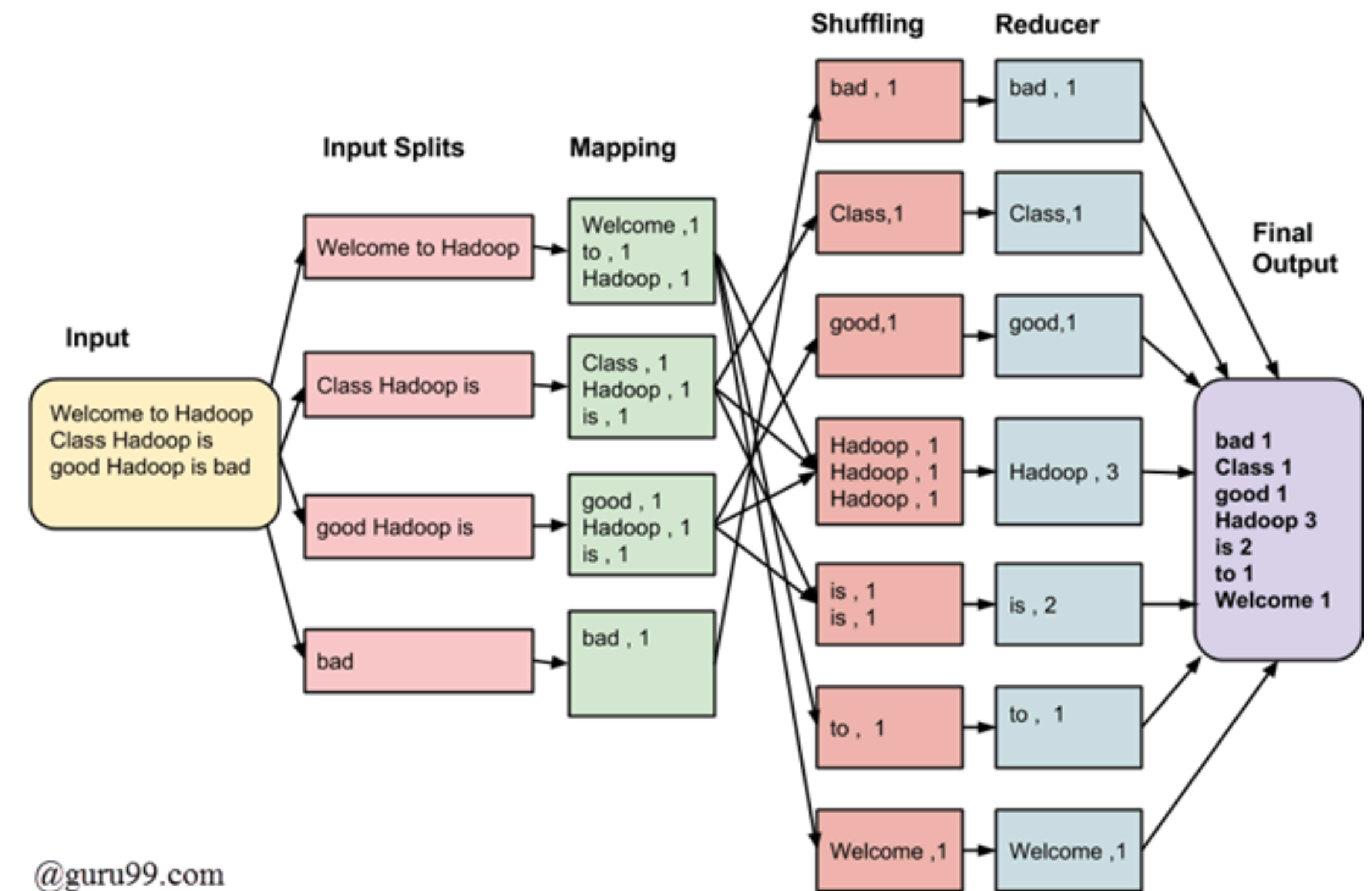
```
SELECT SUM(sales_amount) AS total_sales
FROM employees;
```

In this query, the `SUM()` function is used to add up the values in the "sales\_amount" column for all rows in the "employees" table. The result will be returned as "total\_sales" in the query output.

Remember to replace "employees" with the actual name of your table, and "sales\_amount" with the correct column name in your table that holds the sales data.

# Query Languages for Data: MapReduce Querying

```
db.observations.mapReduce(  
  function map() { ❷  
    var year = this.observationTimestamp.getFullYear();  
    var month = this.observationTimestamp.getMonth() + 1;  
    emit(year + "-" + month, this.numAnimals); ❸  
  },  
  function reduce(key, values) { ❹  
    return Array.sum(values); ❺  
  },  
  {  
    query: { family: "Sharks" }, ❶  
    out: "monthlySharkReport" ❻  
  }  
);
```



# Query Languages for Data: MapReduce Querying

- **MapReduce** is a programming model for processing large amounts of data in bulk across many machines, popularized by Google.
- MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between

```
db.observations.mapReduce(  
  function map() { ②  
    var year = this.observationTimestamp.getFullYear();  
    var month = this.observationTimestamp.getMonth() + 1;  
    emit(year + "-" + month, this.numAnimals); ③  
  },  
  function reduce(key, values) { ④  
    return Array.sum(values); ⑤  
  },  
  {  
    query: { family: "Sharks" }, ①  
    out: "monthlySharkReport" ⑥  
  }  
);
```

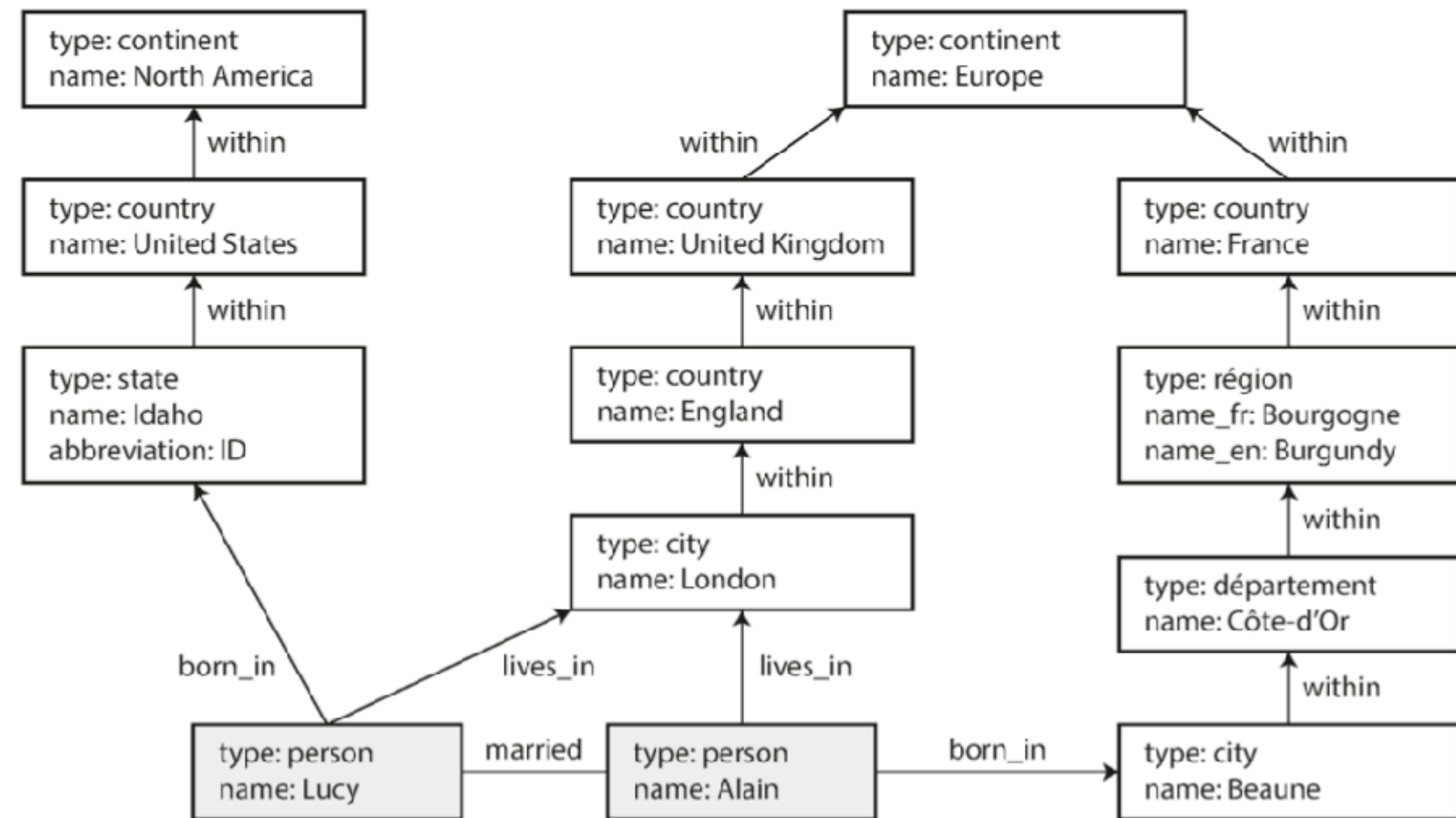


# Graph-Like Data Models

- If your application has mostly one-to-many relationships or no relationships between records, the document model is appropriate.
- A graph consists of two kinds of objects: **vertices** and **edges**.
- Social graphs: Vertices are people, and edges indicate which people know each other.
- The web graph: Vertices are web pages, and edges indicate HTML links to other pages.
- Road or rail networks: Vertices are junctions, and edges represent the roads or railway lines between them.

# Graph-structured data

- Many graph-like algorithms
- Different types of query languages
- Imperative and declarative
- Will discuss more later.



# Graph-Like Data Models: Property Graphs

Each vertex consists of:

- A unique identifier
- A set of outgoing edges
- A set of incoming edges
- A collection of properties (key-value pairs)

Each edge consists of:

- A unique identifier
- The vertex at which the edge starts (the tail vertex)
- The vertex at which the edge ends (the head vertex)
- A label to describe the kind of relationship between the two vertices
- A collection of properties (key-value pairs)

# Property Graphs

- Can store as two tables: one for vertices and one for edges.
- Any vertex can have an edge connecting it with any other vertex. There is no schema that restricts which kinds of things can or cannot be associated.
- Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus **traverse** the graph—i.e., follow a path through a chain of vertices —both forward and backward.
- By using **different labels** for different kinds of relationships, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

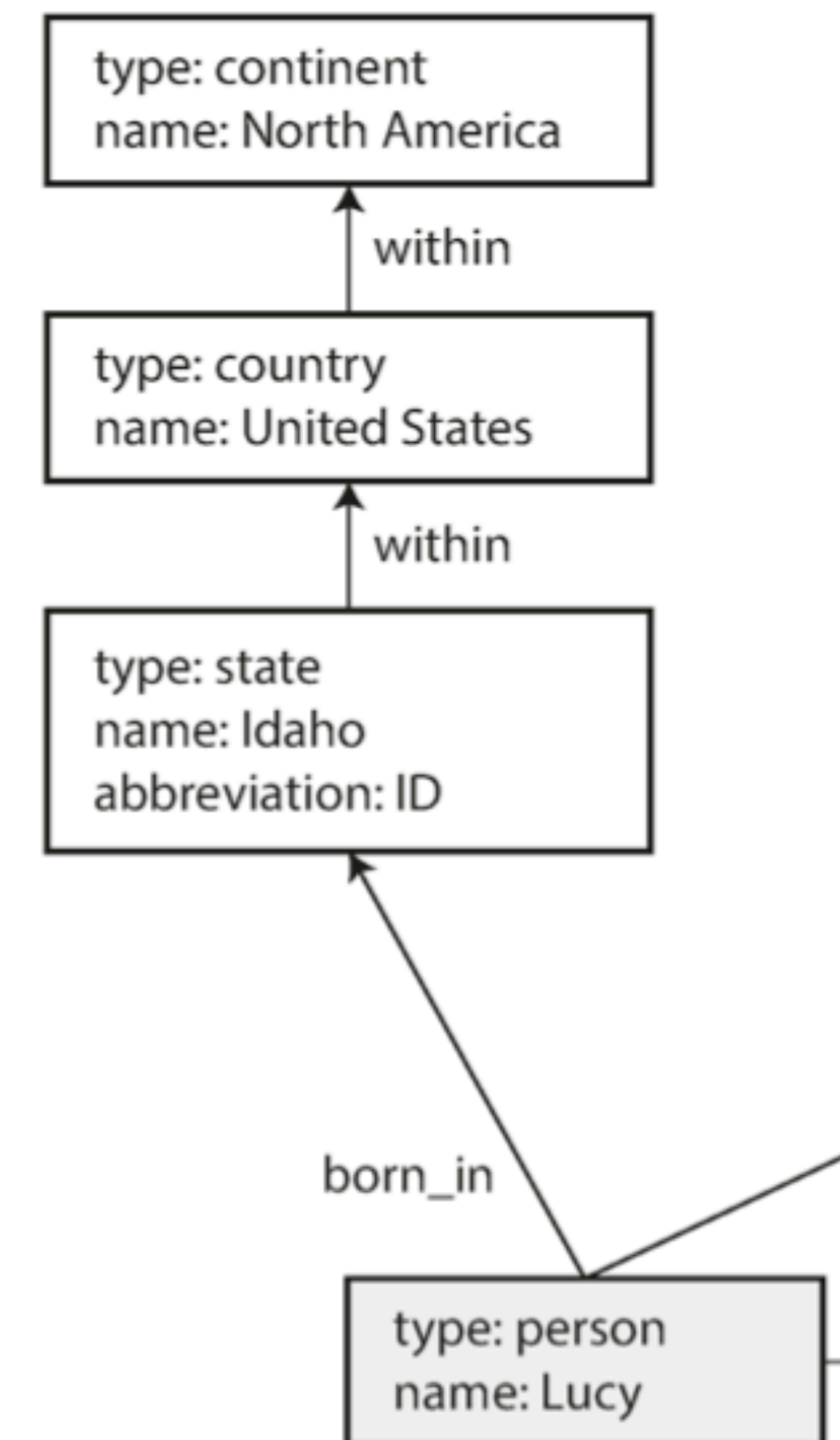


# The Cypher Query Language (Neo4j)

Cypher is a declarative query language for property graphs, created for the graph database.

## CREATE

```
(NAmerica:Location {name:'North America', type:'continent'}),  
(USA:Location      {name:'United States', type:'country'  } ),  
(Idaho:Location    {name:'Idaho',         type:'state'    } ),  
(Lucy:Person       {name:'Lucy' } ),  
(Idaho) -[:WITHIN]-> (USA) -[:WITHIN]-> (NAmerica),  
(Lucy)  -[:BORN_IN]-> (Idaho)
```



# The Cypher Query Language

find people who emigrated from the US to Europe

```
MATCH
  (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

- Multiple ways to execute the query.
  - Person first.
  - Location first.
  - Edge first.

# Triple-Stores and SPARQL

- The triple-store model is mostly equivalent to the property graph model, using different words to describe the same ideas.
- In a triple-store, all information is stored in the form of very simple three-part statements: (**subject**, **predicate**, **object**). The subject of a triple is equivalent to a vertex in a graph.

# Triple-Stores and SPARQL

- The semantic Web
- The Resource Description Framework (RDF) was intended as a mechanism for different websites to publish data in a consistent format, allowing data from different websites to be automatically combined into a web of data—a kind of internet-wide **“database of everything.”**
- SPARQL is a query language for triple-stores using the RDF data model. SPARQL is a nice query language—even if the semantic web never happens, it can be a powerful tool for applications to use internally.

# Today's topic: Data Model Evolutions & Query language

- Object-relational mismatch
- ID v.s. Strings
- Many-to-many v.s. one-to-many
- Hierarchical/network/relational model
- Imperative & declarative Query languages