No SQL Databases

Thomas Schwarz, SJ

Relational Model Shortcomings

- Need: Greater Scalability
  - High write throughput / very large datasets
- Independence from few vendors — Move towards Open Source
- Need for different query operations
- Restrictiveness of relational schemas

Data at Very Large Scale

- Example: Hush — HBase URL Shortener
  - Hand a URL to a Shortener service
  - Get a shorter URL back
    - E.g. to use in twitter messages
  - Shortener provides usage counter for each shortened URLs
  - "Vanity URL" that incorporate specific domain names
  - Need to maintain users
    - log in to create short URLs
    - track existing URLs
    - see reports for daily, weekly, or monthly usage

Data at Very Large Scale

- Data is too large to store at a single server
  - But then:
    - Limited need for transactions
    - Importance of high throughput writes and reads

Data at Very Large Scale

- Columnar Layout
  - A relational database strategy often adopted in No-SQL databases
  - Instead of storing data in tuples
  - Store by attribute

Columnar Layout

- Given a SQL Table:

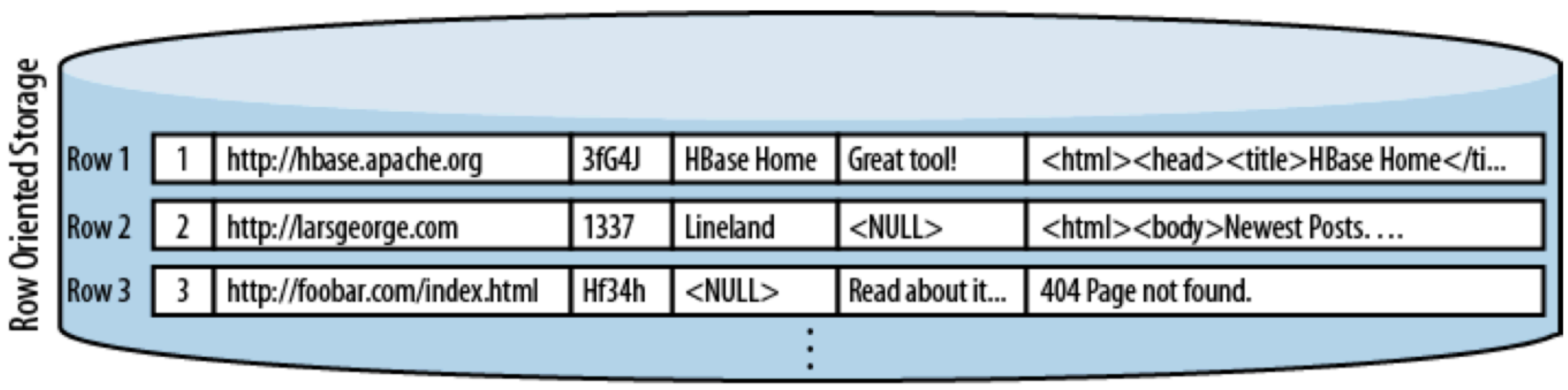
SQL Schema

URLS					
url_id INTEGER PK	**url** VARCHAR(4096)	**ref_short_id** CHAR(8)	**title** VARCHAR(200)	**description** VARCHAR(400)	**content** TEXT
1	http://hbase.apache.org	3fG4J	HBase Home	Great tool!	<html><head><title>HBase Home</ti...
2	http://larsgeorge.com	1337	Lineland	<NULL>	<html><body>Newest Posts...
3	http://foobar.com/index.html	Hf34h	<NULL>	Read about it...	404 Page not found.
4	http://cnn.com/page123.html	Oo0O1	Sport News	Soccer News	<html><body>Results, Reviews, ...

- We project to columns
- We select rows

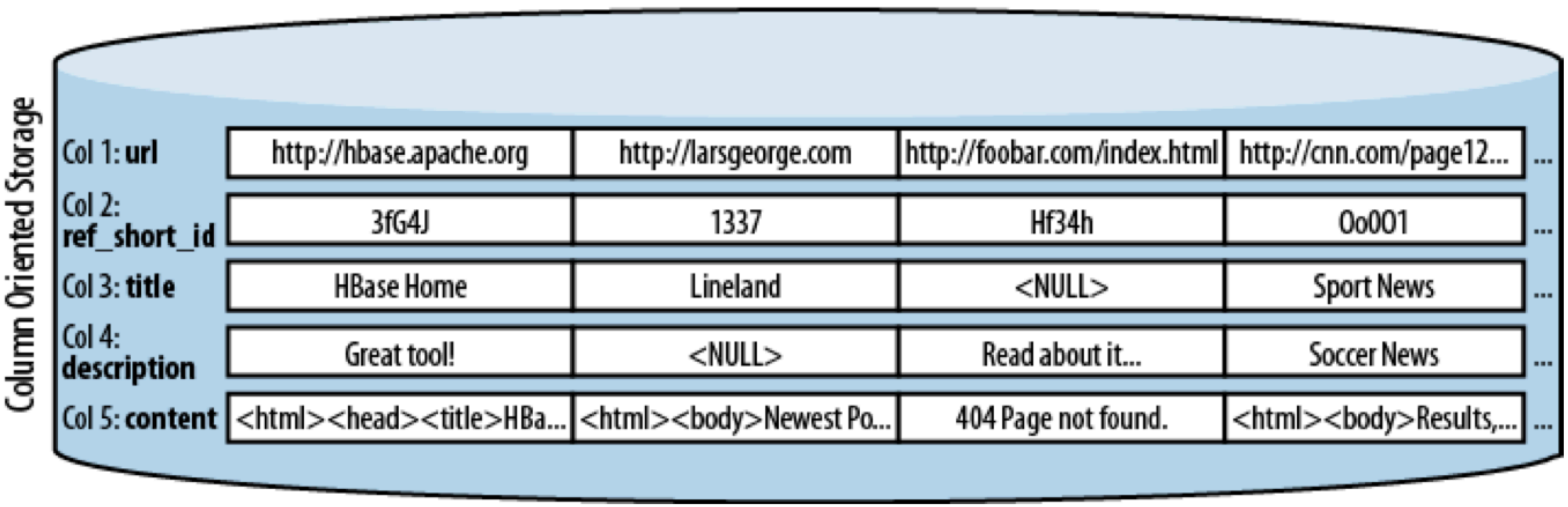
Columnar Layout

- We can store it row-by-row



Columnar Layout

- Or we can use a columnar layout



Data at Very Large Scale

- For large HUSH:
    - Can use a relational database
    - Use normalization and obtain a scheme

Data at Very Large Scale

- To deal with very large data and with high operations volume:
- Principles of Denormalization, Duplication, Intelligent Keys
  - Denormalize by duplicating data in more than one table
    - Avoids aggregation at read time
    - Pre-materialize required views

Data at Very Large Scale

- Denormalization:
  - We normalize to avoid write anomalies
    - A given fact is represented in a single value
    - A new fact or a changed fact affect a single tuple
    - We use joins in order to recombine facts

Data at Very Large Scale

- Example:
    - Orders has OrderNumber, Dates and Status
    - Orderdetails has OrderNumber, Items, Quantities and Prices
    - If a status changes: only update one row in orders

Data at Very Large Scale

- Example continued
  - Price of normalization is joins for a query like:
    - “What is the sales volume by a given sales person?”
  - Denormalization:
    - Join orders and orderdetails on orderNumber
    - Creates a write anomaly: If an order is shipped, need to update several rows
    - But avoids the join
  - You can do this as a materialized view

Data at Very Large Scale

- **D**enormalize, **D**uplication, **I**ntelligent Keys (**DDI**) principles
  - Aggregate related tables into a big table
    - Prefer “tall-narrow” over “flat-wide”
    - I.e. many rows, few columns over many columns, few rows
  - Select the most suitable key: ***row key*** (the intelligent key)
    - Candidates can be measured by the amount of times a primary key becomes a foreign key

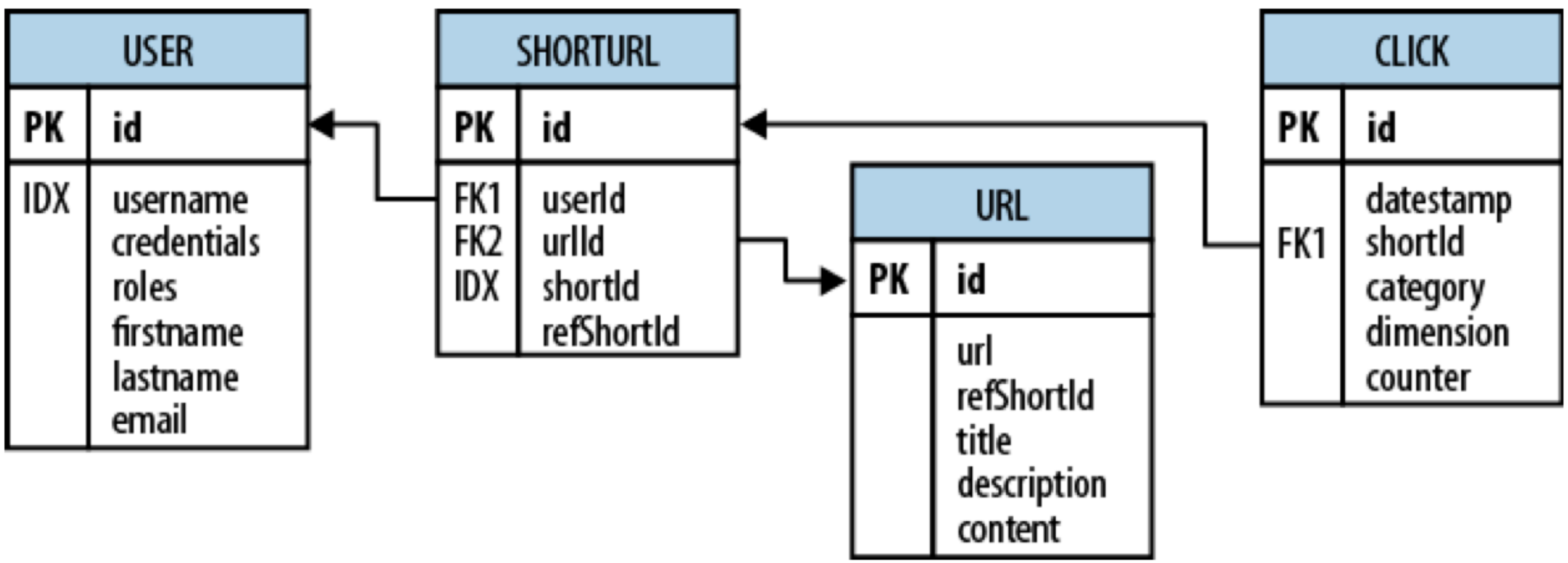
Data at Very Large Scale

- DDI:
  - Once a tall-lean table structure is used
  - Can define ***automatic sharding***
    - Horizontal fragmentation by row-key

Data at Very Large Scale

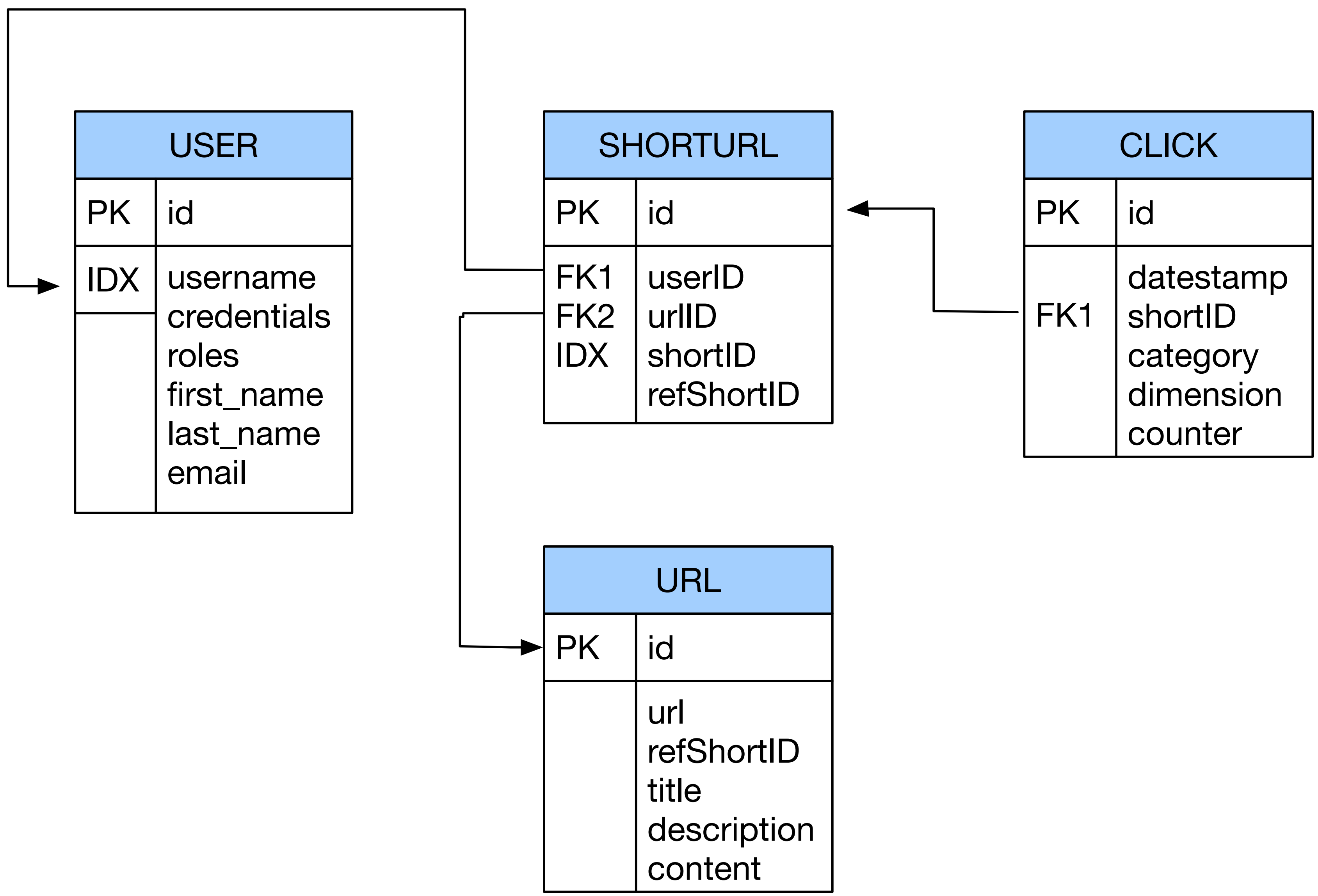
- Example: HBase URL Shortener (Hush)
  - user(<u>id</u>, username, credentials, rules, first_name, last_name, email) with unique username constraint
  - url(<u>id</u>, url, refShortID, title, description, content)
  - shorturl(id, userID, urlID, shortID, refShortID, description) with unique shortID and F.K. userID and urlID
  - click(<u>id</u>, datestamp, shortID, category, dimension, counter) with F.K. shortID

Data at Very Large Scale



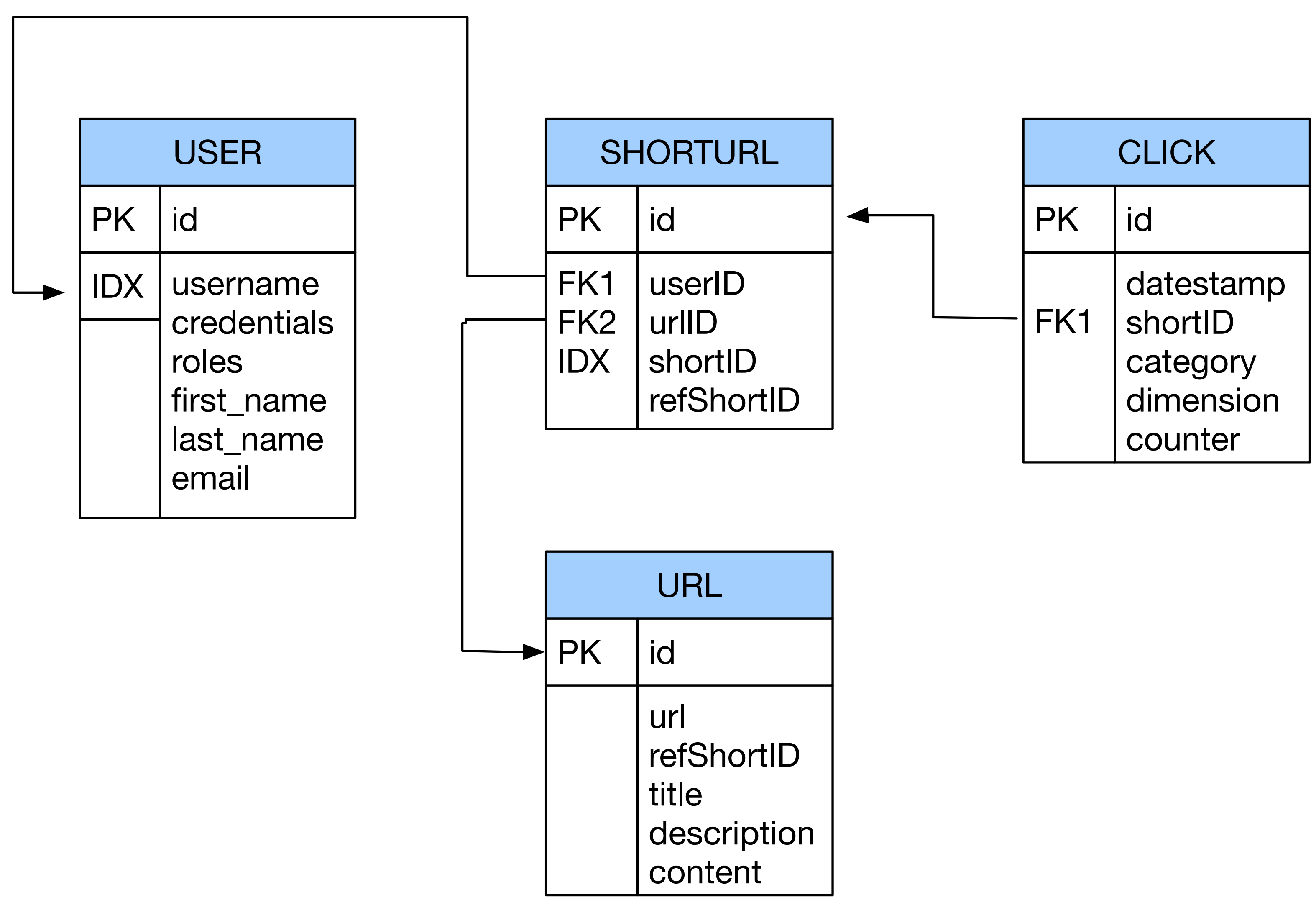
Data at Very Large Scale

- Purpose: maps long URLs to short URLs



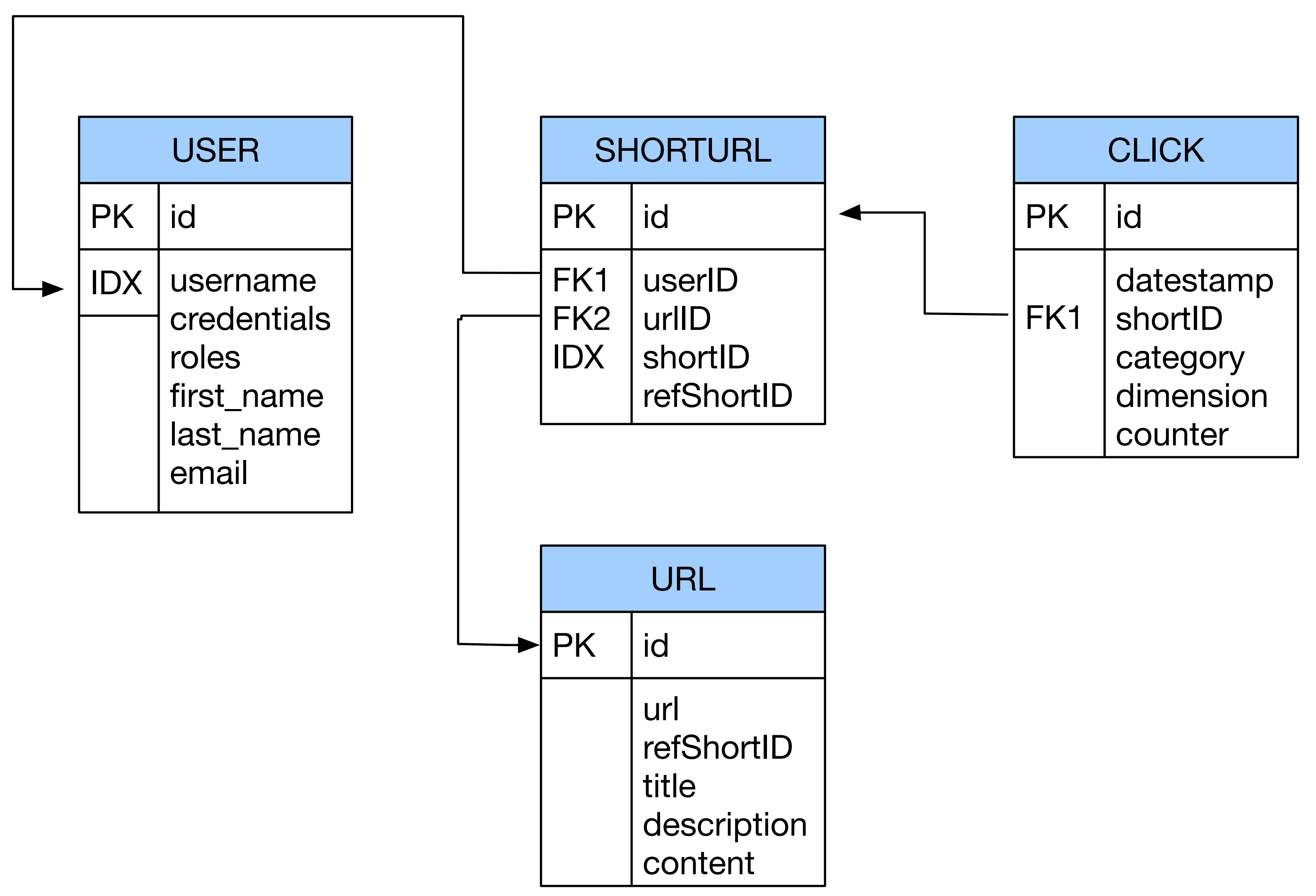
Data at Very Large Scale

- Short URL can be given to others
- This is translated to the full URL



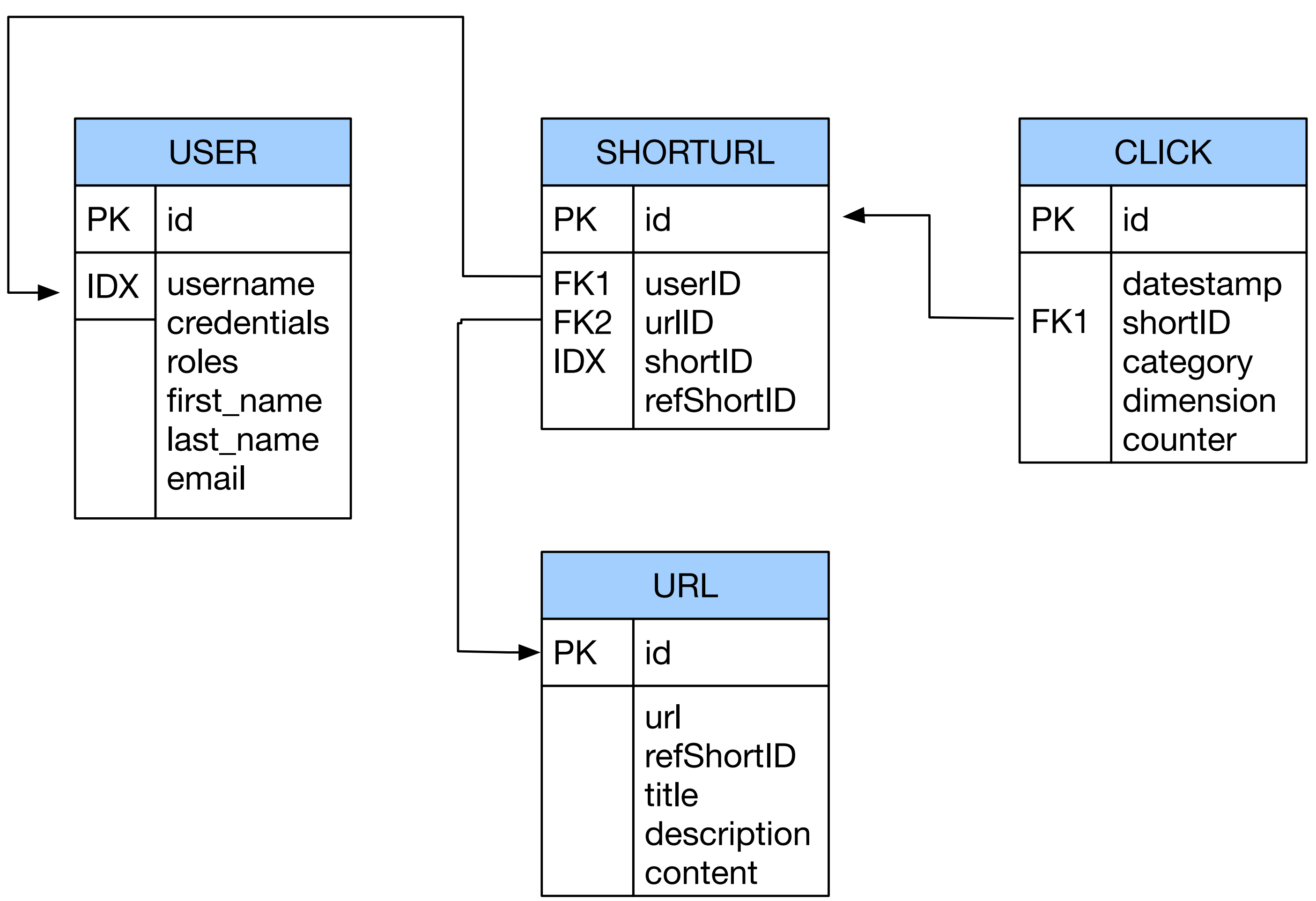
Data at Very Large Scale

- Each click is tracked, which aggregates to weekly usage numbers



Data at Very Large Scale

- All these operations require joins



Data at Very Large Scale

- Bandwidth problem
  - Especially for joins
  - Need to store data in joins together, not look them up separately
  - But can relax on the consistency model:
    - No need to serialize short URL creation or URL translations or have atomic updates
  - Might be able to relax integrity constraints
    - Statistics need to be approximately correct

Data at Very Large Scale

- Denormalization:
  - Key idea: Store data together that is likely to be joined
  - Means:
    - massive duplication of data
    - relaxed consistency needed
    - but faster reads / writes

Fundamental Ideas

- Wide column stores
  - names and format of columns can vary from row to row

<table>
<tr><td rowspan="2">Row 1</td><td>Column 1</td><td>Column 2</td><td>Column 3</td><td>Column 4</td><td rowspan="2">…</td></tr>
<tr><td>Value 1</td><td>Value 2</td><td>Value 3</td><td>Value 4</td></tr>
<tr><td rowspan="2">Row 2</td><td>Column 1</td><td>Column 2</td><td>Column 3</td><td>Column 4</td><td rowspan="2">…</td></tr>
<tr><td>Value 1</td><td>Value 2</td><td>Value 3</td><td>Value 4</td></tr>
<tr><td rowspan="2">Row 3</td><td>Column 1</td><td>Column 2</td><td>Column 3</td><td>Column 4</td><td rowspan="2">…</td></tr>
<tr><td>Value 1</td><td>Value 2</td><td>Value 3</td><td>Value 4</td></tr>
<tr><td rowspan="2">Row 4</td><td>Column 1</td><td>Column 2</td><td>Column 3</td><td>Column 4</td><td rowspan="2">…</td></tr>
<tr><td>Value 1</td><td>Value 2</td><td>Value 3</td><td>Value 4</td></tr>
</table>

…

  - Each row is a key-value pair
  - First implemented with Google's BigTable
  - Implemented by Cassandra, HBase, MS Azure Cosmos DB, …

Fundamental Ideas

- Document databases
  - Uses a format like JSON document
  - MongoDB, XML databases

Fundamental Ideas

- Key-value database
  - Every record is a key-value pair
  - A large set of tools predating no-sql databases in general

Fundamental Ideas

- Graph databases
  - Navigational database successor:
    - information about data interconnectivity or topology as important as data itself
  - See below for an example

NoSQL Consistency

- Consistency Levels for NoSQL databases
  - Strict: changes to data are atomic and (appear to) take effect immediately
  - Sequential: every client sees all changes in the same order in which they are applied
  - Causal: all changes that are casually related are observed in the same order by all clients
  - Eventual: When no updates occur for a while, then all pending updates will occur and all replicas are consistent
  - Weak: No guarantee is made

CAP Theorem

- A distributed system can only achieve two out of the three goals of
  - Consistency
  - Availability
  - Partition Tolerance

A. Fox and E. A. Brewer, "Harvest Yield and Scalable Tolerant Systems", *Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99) IEEE CS*, pp. 174-178, 1999.

Brewer, Eric. "CAP twelve years later: How the" rules" have changed." Computer 45.2 (2012): 23-29.

Example: HURL

- HBase:

Table: shorturl		
Row Key:	shortId	
Family:	data:	Columns: url, refShortId, userId, clicks
	stats-daily: **[ttl: 7days]**	Columns: YYYYMMDD, YYYYMMDD\x00<country-code>
	stats-weekly: **[ttl: 4weeks]**	Columns: YYYYWW, YYYYWW\x00<country-code>
	stats-monthly: **[ttl: 12months]**	Columns: YYYYMM, YYYYMM\x00<country-code>

Table: url		
Row Key:	MD5(url)	
Family:	data: **[compressed]**	Columns: refShortId, title, description
	content: **[compressed]**	Columns: raw

Table: user-shorturl		
Row Key:	username\x00shortId	
Family:	data:	Columns: timestamp

Table: user		
Row Key:	username	
Family:	data:	Columns: credentials, roles, firstname, lastname, email

Alternatives to Relational Schemes: XML

- Data is often structured hierarchically

```
Invoice = {
  date : "2008-05-24"
  invoiceNumber : 421

  InvoiceItems : {
    Item : {
      description : "Wool Paddock Shet Ret Double Bound Yellow 4'0"
      quantity : 1
      unitPrice : 105.00
    }
    Item : {
      description : "Wool Race Roller and Breastplate Red Double"
      quantity : 1
      unitPrice : 75.00
    }
    Item : {
      description : "Paddock Jacket Red Size Medium Inc Embroidery"
      quantity : 2
      unitPrice : 67.50
    }
  }
}
```

Alternatives to Relational Schemes: XML

- As an XML document

```xml
<invoice>

 <number>421</number>
 <date>2008-05-24</date>
 <items>
  <item>
   <description>Wool Paddock Shet Ret Double Bound Yellow 4'0"</description>
   <quantity>1</quantity>
   <unitPrice>105.00</unitPrice>
  </item>
  <item>
   <description>Wool Race Roller and Breastplate Red Double</description>
   <quantity>1</quantity>
   <unitPrice>75.00</unitPrice>
  </item>
  <item>
   <description>Paddock Jacket Red Size Medium Inc Embroidery</description>
   <quantity>2</quantity>
   <unitPrice>67.50</unitPrice>
  </item>
 </items>
</invoice>
```

Alternatives to Relational Schemes: XML

- Advantage of XML
  - Faster to scan all data
  - No joins
- Disadvantages of XML
  - Each record contains the full or an abbreviated scheme
  - Each query needs to select from big chunks of data

Alternatives to Relational Schemes: JSON

- JSON — JavaScript Object Notation
  - Human-readable
  - Organized as key-value pairs

Alternatives to Relational Schemes: JSON

- JSON record example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Alternatives to Relational Schemes: JSON

- JSON can use a schema (type definition)
- JSON was first used for data transmission as a data serialization format

Alternatives to Relational Schemes: JSON

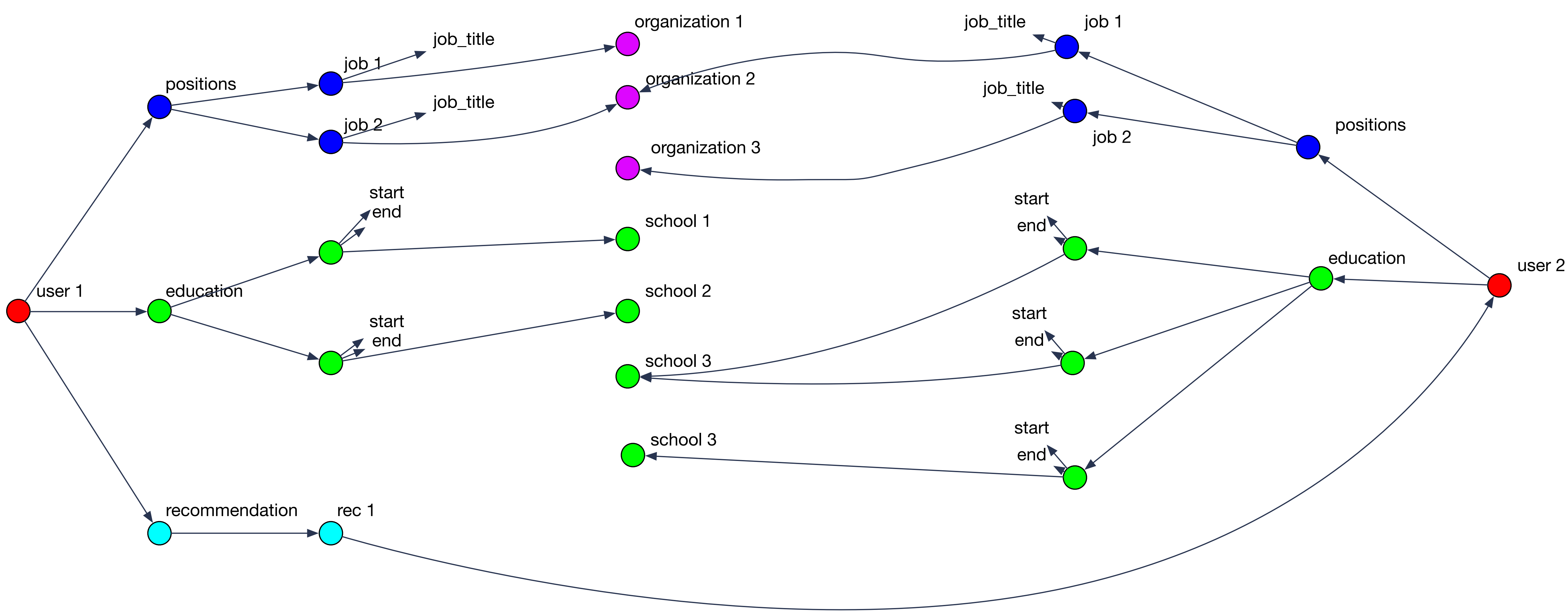
- Many-to-One and Many-to-Many Relationships
  - Modeled by the same value for the same key
    - Problem: Need to standardize / internationalize these values
    - Using id-s instead of plain text to avoid problems
    - Table of id-s reintroduce a relational scheme through a backdoor

Alternatives to Relational Schemes: JSON

- Resumé
  - Users present people
  - People have jobs, education, and recommenders
- But they share jobs, companies, degrees, schools, recommenders
  - Should they stay text strings or become entities?
    - Latter allows to add information to all resumés
- If recommenders get a photo, then all resumés should be updated with this photo, so better to make recommenders entities

Alternatives to Relational Schemes: JSON

- Data has a tendency to become less-join free



Document Databases

- Records are documents
  - Encode in
    - XML
    - YAML
    - JSON
    - BSON (Mongo DB)
  - CRUD operations: create, read, update, delete

Document Databases

- Enforcing schema
  - Most document databases do not enforce schema
    - —> “Schemaless”
    - In reality: “Schema on Read”
    - RDBMS would then use “Schema on Write”
- Allows schema updates in simple form

Document Databases

- Schema on Read:
  - Advantages:
    - Data might come from external sources
  - Disadvantages:
    - No data checking

Document Databases

- Document database support
  - Most commercial database systems now support XML databases

Map Reduce

Data at Scale

History

- A *simple* paradigm that popped up several times as paradigm
- Observed by google as a software pattern:
  - Data gets filtered locally and filtered data is then reassembled elsewhere
  - Software pattern: Many engineers are re-engineering the same steps
- Map-reduce:
  - Engineer the common steps efficiently
  - Individual problems only need to be engineered for what makes them different

History

- Open source project (in part sponsored by Yahoo!)
  - Java-based Hadoop
  - Eventually a first tier Apache Foundation project

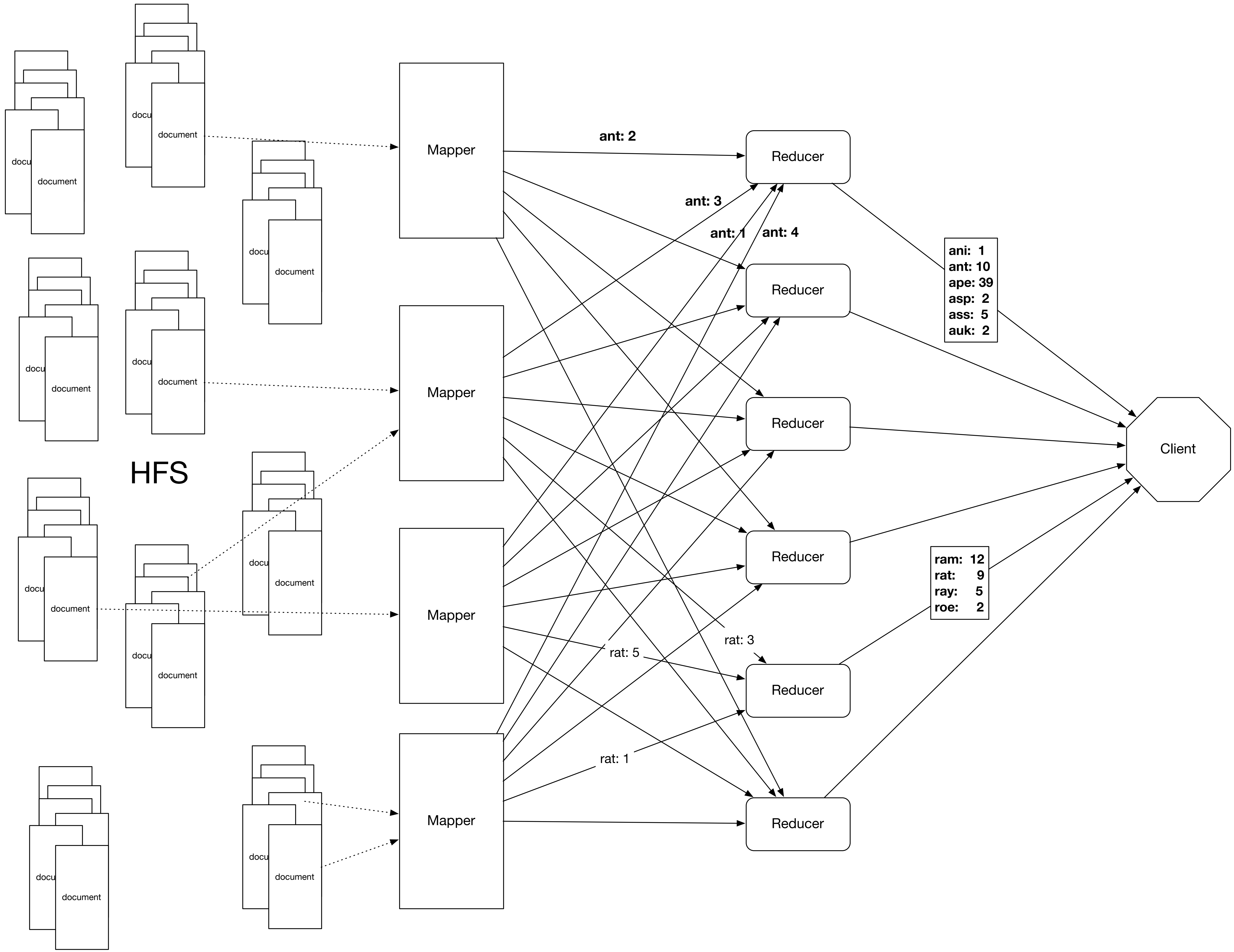
- Other projects at higher level: Pig, Hive, HBase, Mahout, Zookeeper
  - Use Hadoop as foundation
  - Hadoop is becoming a distributed OS

Map Reduce Paradigm

- Input: Large amount of data spread over many different nodes
- Output: A single file of results
- Two important phases:
  - **Mapper**: Records are processed into key-value pairs. Mapper sends key-value pairs to reducers
  - **Reducer:** Create final answer from mapper

Simple Example

- Hadoop Word Count
  - Given different documents on many different sites
    - Mapper:
      - Extract words from record
      - Combines words and generates key-value pairs of type word: key
      - Sends to the reducers based on hash of key
    - Reducer:
      - Receives key-value pairs
      - Adds values for each key
      - Sends accumulated results to aggregator - client



HFS
document
docu
Mapper
Reducer
Client
ant: 2
ant: 3
ant: 1
ant: 4
rat: 5
rat: 3
rat: 1
ani: 1
ant: 10
ape: 39
asp: 2
ass: 5
auk: 2
ram: 12
rat: 9
ray: 5
roe: 2

Map-reduce paradigm in detail

- The simple mapper -reducer paradigm can be expanded into several, typical components

Map Reduce in Detail

- **Mapper:**
  - Record Reader
    - Parses the data into records
    - Example: Stackoverflow comments.
      - ```
        <row Id="5" PostId="5" Score="2" Text="Programming in
        Portland, cooking in Chippewa ; it makes sense that these
        would be unlocalized. But does bicycling.se need to follow
        only that path? I agree that route a to b in city x is not
        a good use of this site; but general resources would be."
        CreationDate="2010-08-25T21:21:03.233" UserId="21" />
        ```
    - Record reader extract the “Text=” string
    - Passes record into a key-value format to rest of mapper
        ```

Map Reduce in Detail

- **Mapper**
  - map
    - Produces “intermediate” key-value pairs from the record
    - Example:
      - **"Programming in Portland, cooking in Chippewa ; it makes sense that these would be unlocalized. But does bicycling.se need to follow only that path? I agree that route a to b in city x is not a good use of this site; but general resources would be.”**
      - Map produces: <programming: 1> <in: 1> <Portland: 1> <cooking: 1> <in: 1> …

Map Reduce in Detail

- **Mapper**
    - Combiner — a local reducer
        - Takes key-value pairs and processes them
        - Example:
            - Map produces: <programming: 1> <in: 1> <Portland: 1> <cooking: 1> <in: 1> …
            - Combiner combines words: <programming: 1> <in: 4> <Portland: 3> …

Map Reduce in Detail

- Combiners allow us to reduce network traffic
  - By compacting the same infomrmation

Map Reduce in Detail

- **Mapper**
  - Partitioner
    - Partitioner creates shards of the key-value pairs produced
    - One for each reducer
    - Often uses a hash function or a range
    - Example:
      - md5(key) mod (#reducers)

Map Reduce in Detail

- **Reducer**
  - Shuffle and Sort
  - **Part of the map-reduce framework**
    - Incoming key-value pairs are sorted by key into one large data list
    - Groups keys together for easy agglomeration
    - Programmer can specify the comparator, but nothing else

Map Reduce in Detail

- **Reducer**
  - reduce
    - Written by programmer
    - Works on each key group
    - Data can be combined, filtered, aggregated
    - Output is prepared

Map Reduce in Detail

- **Reducer**
  - Output format
    - Formats final key-value pair

Map Reduce Patterns

- Summarizations
  - Input: A large data set that can be grouped according to various criteria
  - Output: A numerical summary
  - Example:
    - Calculate minimum, maximum, total of certain fields in documents in xml format ordered by user-id

Summarization

- Example:
    - Given a database in xml-document format


```
<row Id="193" PostTypeId="1" AcceptedAnswerId="194"
CreationDate="2010-10-23T20:08:39.740" Score="3" ViewCount="30"
Body="&lt;p&gt;Do you lose one point of reputation when you
down vote community wiki?  Meta? &lt;/p&gt;&#xA;&#xA;&lt;p&gt;I
know that you do for &quot;regular questions&quot;. &lt;/
p&gt;&#xA;" OwnerUserId="134"
LastActivityDate="2010-10-24T05:41:48.760" Title="Do you lose
one point of reputation when you down vote community wiki?
Meta?" Tags="&lt;discussion&gt;" AnswerCount="1"
CommentCount="0" />
```


    - Determine the earliest, latest, and number of posts for each user

Summarization

- Mapper:
  - Step 1: Preprocess document by extracting the user ID and the date of the post
  - Step 2: map:
    - User ID becomes the key.
    - Value stores the date twice in Java-date format and adds a long value of 1

```
“134”:  (2010-10-23T20:08:39.740, 2010-10-23T20:08:39.740, 1)
```

Summarization

- Mapper:
  - Step 3: Combiner
    - Take intermediate User-ID — value pairs
    - Combine the value pairs
      - Combination of two values:
        - first item is minimum of the dates
        - second item is maximum of the dates
        - third item is sum of third items

Summarization

- The map reduce framework is given the number of reducers
  - Autonomously maps combiner results to reducers
  - Each reducer gets key-value parts for a range of user-IDs grouped by user-ID

Summarization

- Reducer:
  - Passes through each group combining key-value pairs
  - End-result:
    - Key-value pair with key = user-id
    - Value is a triple with
      - minimum posting date
      - maximum posting date
      - number of posts

Summarization

- Reducer:
    - Each summary key— value pair is sent to client

Summarization

- Example (cont.)

Mapper 1

UserID 12345	01.02.2010	01.02.2010	1
UserID 12345	02.02.2010	02.02.2010	1
UserID 12345	04.02.2010	04.02.2010	1
UserID 98765	12.02.2010	12.02.2010	1
UserID 98765	02.02.2010	02.02.2010	1
UserID 98765	05.02.2010	05.02.2010	1
UserID 56565	02.02.2010	02.02.2010	1
UserID 56565	03.02.2010	03.02.2010	1

Combiner →

UserID 12345	01.02.2010	04.02.2010	3
UserID 98765	02.02.2010	12.02.2010	3
UserID 56565	02.02.2010	03.02.2010	2

Mapper 2

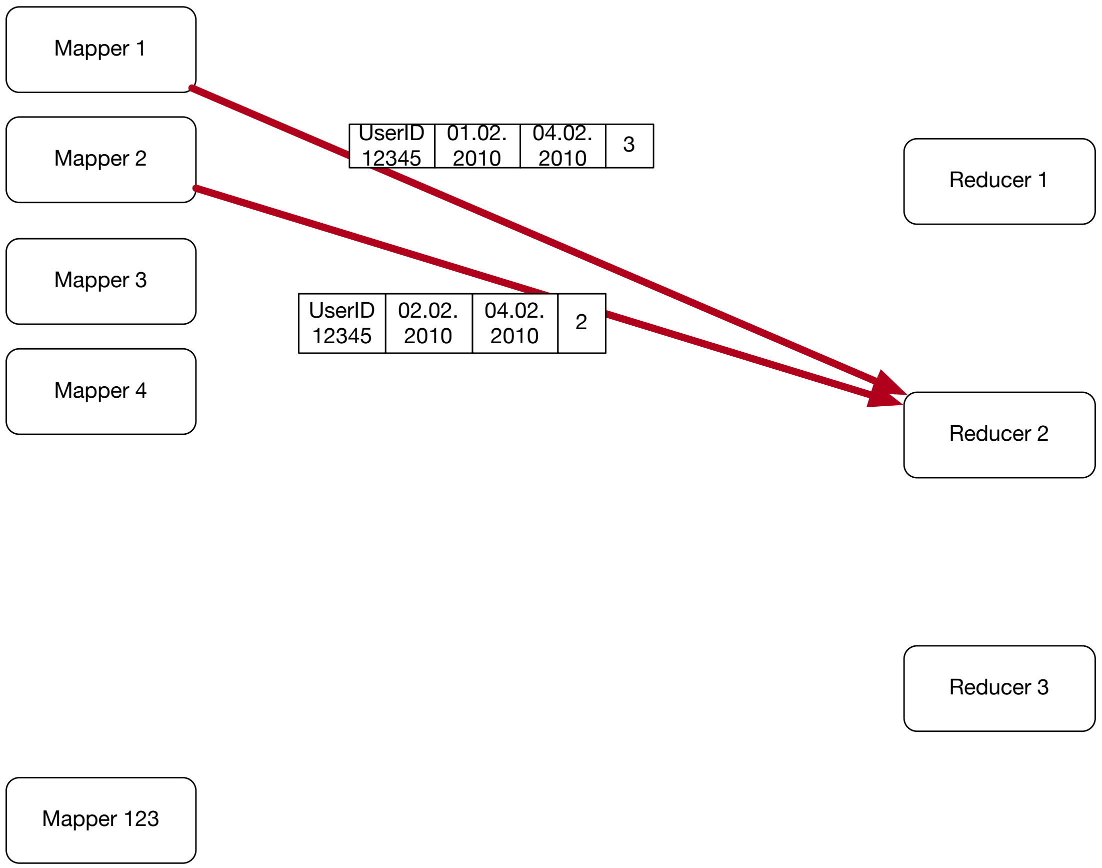
UserID 12345	02.02.2010	02.02.2010	1
UserID 12345	04.02.2010	04.02.2010	1
UserID 77444	12.02.2010	12.02.2010	1
UserID 77444	02.02.2010	02.02.2010	1
UserID 98765	05.02.2010	05.02.2010	1

Combiner →

UserID 12345	02.02.2010	04.02.2010	2
UserID 77444	02.02.2010	12.02.2010	2
UserID 98765	05.02.2010	05.02.2010	1

Summarization

- Example (cont.) Automatic Shuffle and Sort
  - Records with the same key are sent to the same reducer



Summarization

- Example (cont.)
  - Reducer receives records already ordered by user-ID
  - Combines records with same key

UserID 12345	01.02.2010	04.02.2010	3
UserID 12345	02.02.2010	04.02.2010	2
UserID 12345	26.03.2010	30.04.2010	5
UserID 12345	19.01.2010	01.04.2010	3
UserID 16542	02.02.2010	04.02.2010	6
UserID 16542	26.03.2010	29.05.2010	5
UserID 16542	19.01.2010	19.01.2010	1

→

UserID 12345	01.02.2010	30.02.2010	13
UserID 16542	19.01.2010	29.05.2010	12

Summarization

- In (pseudo-)pig:
  - Load data

```
posts = LOAD '/stackexchange/posts.tsv.gz'
USING PigStorage('\t') AS (
post_id : long,
user_id : int,
text : chararray,
…
post : date
)
```

Summarization

- In (pseudo-)pig:
  - Group by user-id

```
post_group = GROUP posts BY user_id;
```

  - Obtain min, max, count:

```
result = FOREACH post_group GENERATE group,
MIN(posts.date), MAX(posts.date),
COUNT_STAR(post_group)
```

Summarization

- In (pseudo-)pig:
  - Load data

```
orders = LOAD '/stackexchange/posts.tsv.gz'
USING PigStorage('\t') AS (
post_id : long,
user_id : int,
text : chararray,
…
post : date
)
```

Summarization

- Your turn:
  - Calculate the average score per user
  - The score is kept in the “score”-field

Summarization

- Solution:
  - Need to aggregate sum of score and number of posts
  - Mapper: for each user-id, create a record with score

    `userid: score, 1`

  - Combiner adds scores and counts

    `userid: sum_score, count`

  - Reducer combines as well
  - Generates output key-value pair and sends it to the user
  - `userid: sum_score/count`

Summarization

- Finding the median of a numerical variable
  - Mapper aggregates all values in a list
  - Reducer aggregates all values in a list
  - Reducer then determines median of the list
- Can easily run into memory problems

Summarization

- Median calculation:
  - Can compress lists by using counts
    - `2, 3, 3, 3, 2, 4, 5, 2, 1, 2` becomes

      `(1,1), (2,4), (3,3), (4,1) (5,1)`

    - Combiner creates compressed lists
    - Reducer code directly calculates median
      - An instance where combiner and reducer use different code

Summarization

- Standard Deviation
  - Square-root of variance
  - Variance — Average square deviation from average
  - $$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$
    - Leads to a two pass solution, calculate average first

Summarization

- Standard Deviation
  - Numerically dangerous one-path solution
  - $$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$
    $$= \frac{1}{N}\sum_{i=1}^{N}(x_i^2 - 2\bar{x}x_i + \bar{x}^2)$$
    $$= \frac{1}{N}\sum_{i=1}^{N}x_i^2 - 2\bar{x}\frac{1}{N}\sum_{i=1}^{N}x_i + \bar{x}^2$$
    $$= \frac{1}{N}\sum_{i=1}^{N}x_i^2 - 2\bar{x}^2 + \bar{x}^2 = \frac{1}{N}\sum_{i=1}^{N}x_i^2 + \bar{x}^2$$

Summarization

- Chan's adaptation of Welford's online algorithm
  - Using the counts of elements, can calculate the variance in parallel from any number of partitions

```
def parallel_variance(avg_a, count_a, var_a, avg_b, count_b, var_b):
    delta = avg_b - avg_a
    m_a = var_a * (count_a - 1)
    m_b = var_b * (count_b - 1)
    M2 = m_a + m_b + delta ** 2 * count_a * count_b / (count_a + count_b)
    return M2 / (count_a + count_b - 1)
```

  - Unfortunately, can still be numerically instable

Summarization

- Standard Deviation:
  - Schubert & Gertz: Numerically Stable Parallel Computation of (Co)-Variance
    - SSDBM '18 Proceedings of the 30th International Conference on Scientific and Statistical Database Management

Summarization

- Inverted Index
  - Analyze each comment in StackOverflow to find hyperlinks to Wikipedia
  - Create an index of wikipedia pages pointing to StackOverflow comments that link to them

Summarization

- Inverted Index is a group-by problem solved almost entirely in the map-reduce framework

Summarization / Inverted Index

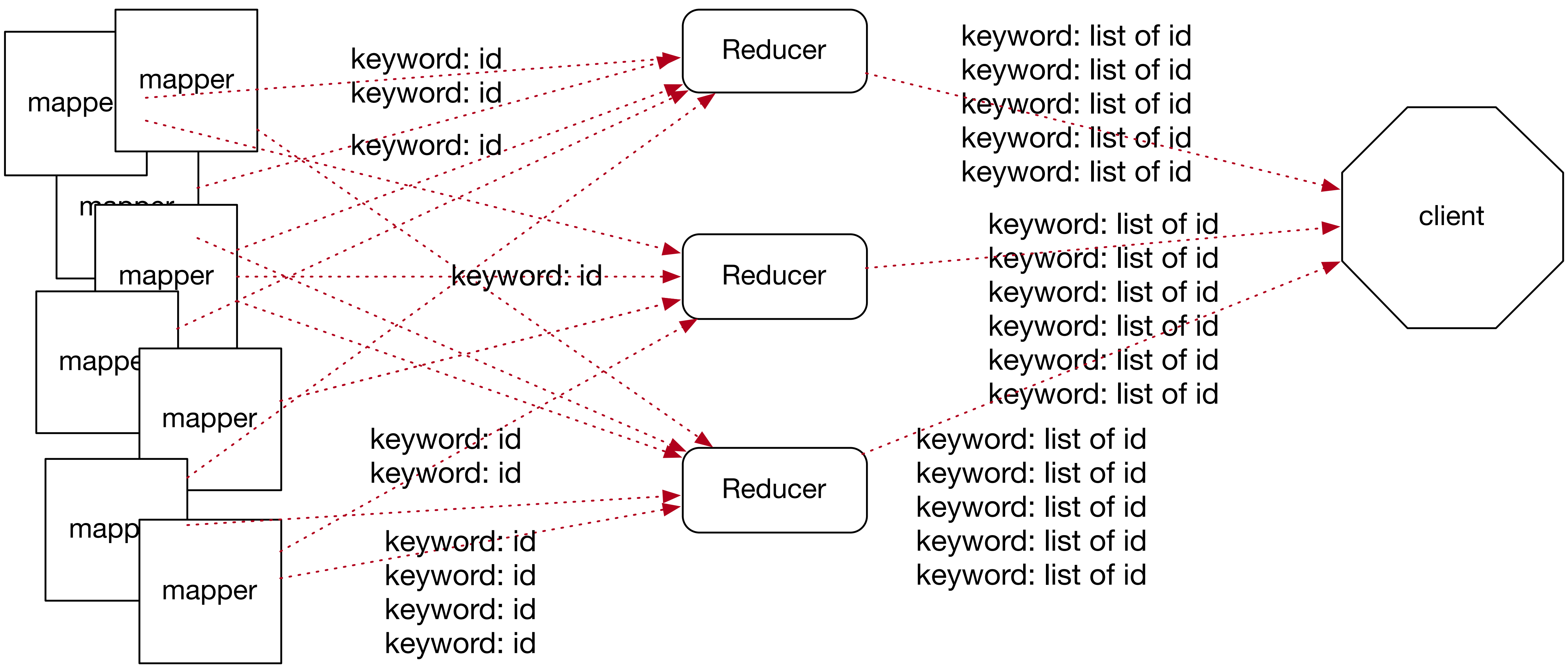
- **Mapper**
  - Parser:
    - Processes posts
    - Checks for right type of post, extracts a list of wikipedia urls (or Null if there are none)
    - Outputs key-value pairs :
      - Keys: wikipedia url
      - Value: row-ID of post
    - Optional combiner:
      - Aggregates values for a wikipedia url in a single list

Summarization / Inverted Index

- **Reducer**
  - Aggregates values belonging to the same key in a list

Summarization / Inverted Index

- Generic Inverted Index diagram



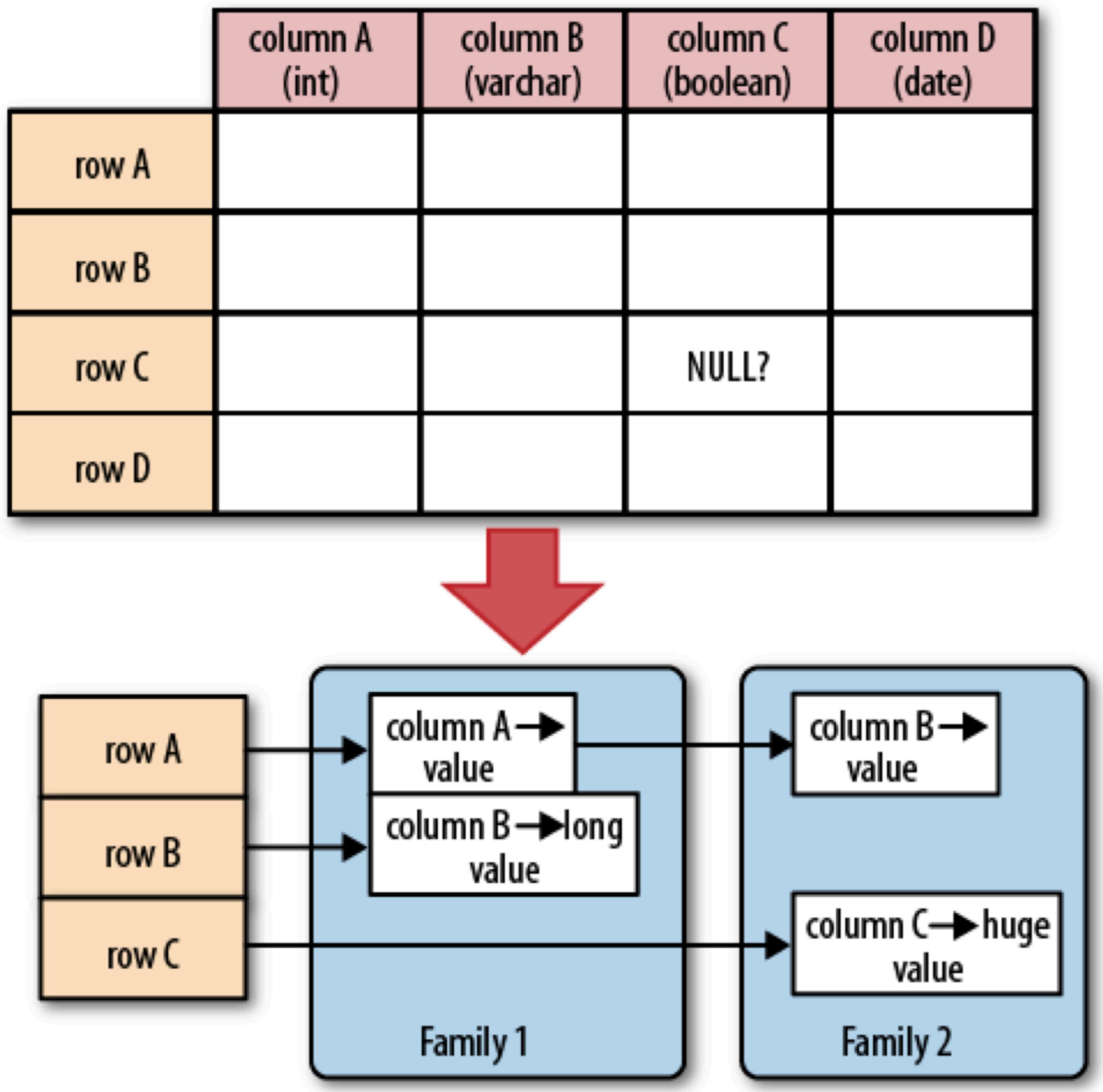
HBase

- Based on BigTable (Google 2006)
- Uses ideas of Map-reduce and Google File System (replication)

HBase

- Basic unit is a column (value)
  - Each *column* can have different versions, each stored in a separate *cell*
- Columns form *rows* (with row identifier)
  - Groups of columns are formed in *families*
  - Columns accessed by *family : qualifier* pairs
- Rows are sorted lexicographically by row key

HBase


column A
(int)
column B
(varchar)
column C
(boolean)
column D
(date)
row A
row B
row C
row D
NULL?
row A
row B
row C
column A→
value
column B→long
value
column B→
value
column C→huge
value
Family 1
Family 2

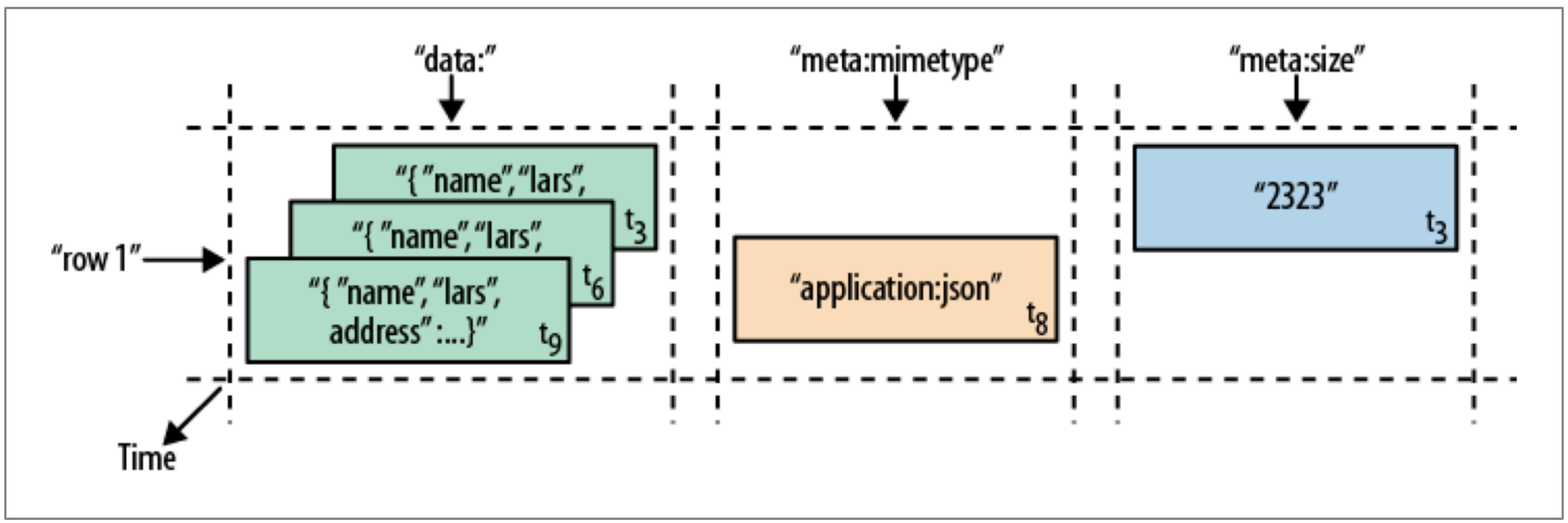
HBase

- Conceptually:
  - HBase table looks like a relational database table
  - But access is organized via *tags*:

```
(Table, RowKey, Family, Column, Timestamp) → Value
```

HBase

- API allows you to filter data based on conditions on the time



HBase

- Canonical example is the WebTable:
  - Pages obtained crawling the internet
    - Row key is the URL (in reverse order)
      - Contents family: HTTP
      - Anchor family: out-going urls
- Time dimension allows to find pages that are updated frequently

HBase

- Actual storage is in regions
  - A region is a contiguous collection of rows and columns
  - If a region becomes to big: Split it around a middle row key
- Typical region size is a few GBs
- Each server should have between 100 and 10,000 regions

HBase

- API:
  - Allows creation / deletion of tables
  - Allows CRUD access
  - Scan API
  - Supports single-row transactions, but not cross-row transactions
  - Map-reduce framework allows to use tables as input sources

HBase

- Storage implementation:
  - Data is stored in HFiles
    - HFiles have a block index:
      - Can find a row in an HFile with a single disk seek
    - HFiles are immutable
  - Files are stored in the Hadoop Distributed File System (HDFS)

HBase

- To delete a value:
  - Need to use a delete marker with the key (*tombstone marker*)
  - Periodically: Go through HFiles are rewrite them, leaving out deleted rows

Query Languages

- Documents lend themselves to object-oriented querying
  - Imperative code
- SQL is declarative:
  - Programmer explains a solution
  - System figures out the best way to find the solution
- Use declarative query languages for document databases

Query Languages

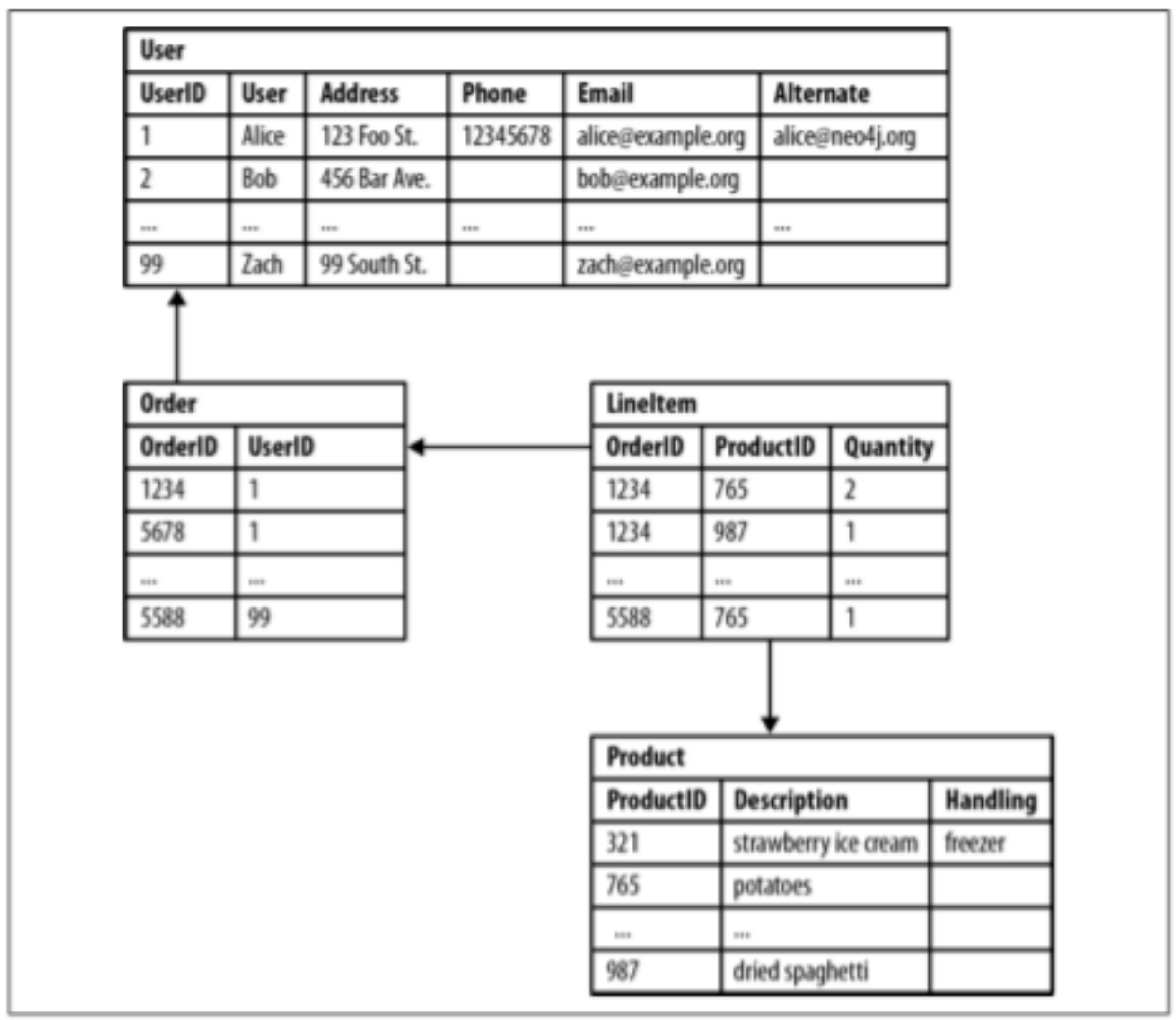
- Map-Reduce (neither declarative nor imperative):
  - Consists of only two pieces of code
    - Mapping: Selecting from Documents
    - Reducing: Take selection elements and operate on them

Alternatives to Relational Schemes: Graph Models

- Graphs consists of vertices and edges
  - Example:
    - Social graphs: vertices are people and edges are relationships such “knows”
    - Web graph: vertices are pages and edges are links
    - Road networks: vertices are places and edges are connections

Alternatives to Relational Schemes: Graph Models

- Relational Database hides semantic relationships



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

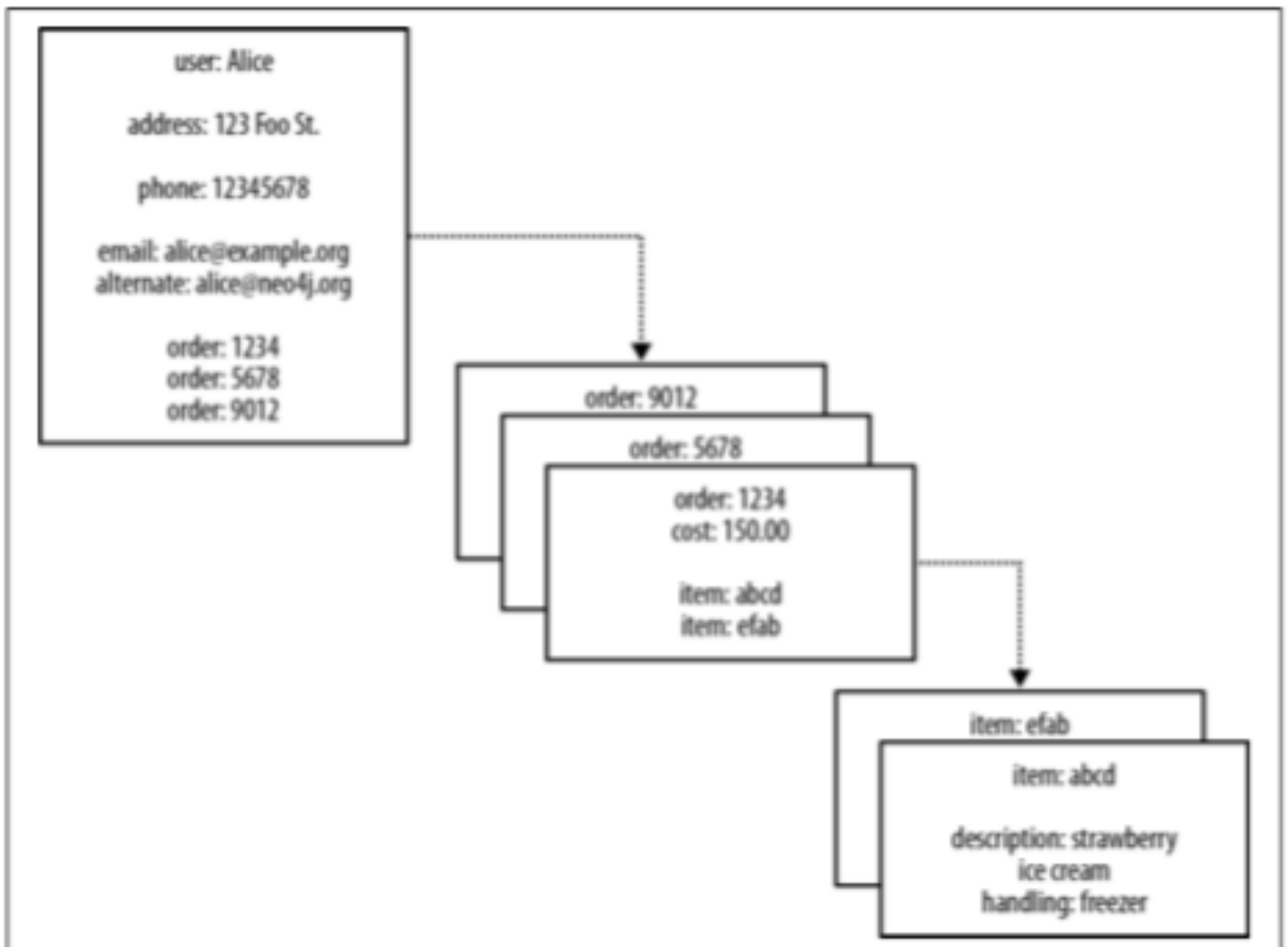
OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

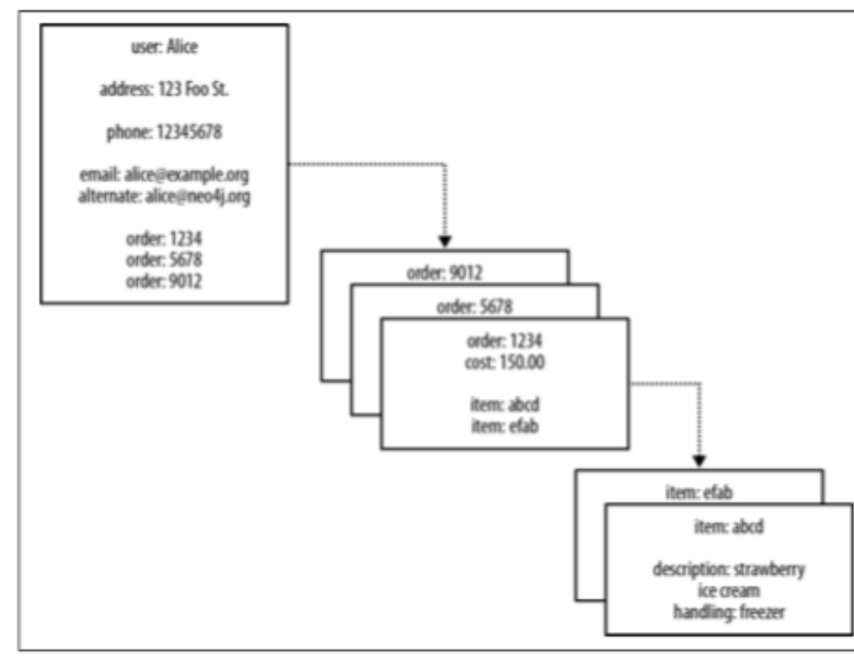
Alternatives to Relational Schemes: Graph Models

- Document model hides semantic relationships



Alternatives to Relational Schemes: Graph Models

- Some property values are really references to foreign aggregates
  - Aggregate’s identifier is a foreign key
- Relationships between them are not explicitly accessible
  - Joining aggregates becomes expensive



Alternatives to Relational Schemes: Graph Models

- Relational Database
  - Some queries are simple:

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
ON PersonFriend.FriendID = p1.ID JOIN Person p2
ON PersonFriend.PersonID = p2.ID WHERE p2.Person = 'Bob'
```

Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

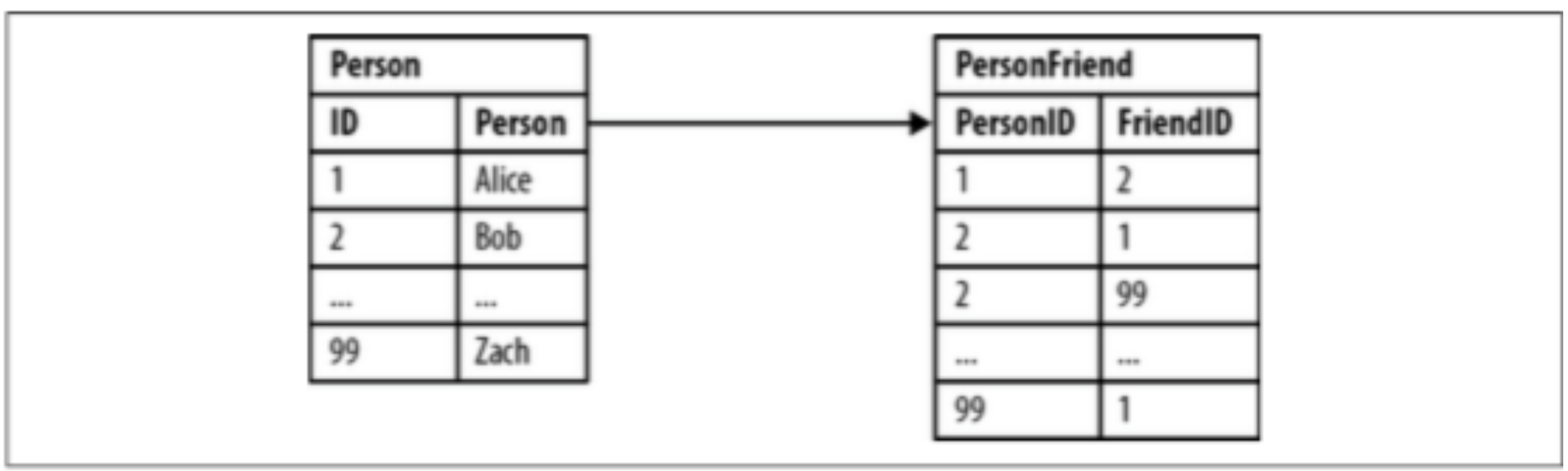
PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Alternatives to Relational Schemes: Graph Models

- Relational Database
  - Some queries are more involved: Friends of Bob

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
   ON PersonFriend.PersonID = p1.ID JOIN Person p2
   ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Alternatives to Relational Schemes: Graph Models

- Relational Database
  - Some queries others are difficult: Alice’s friends of friends

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND FROM
PersonFriend pf1 JOIN Person p1
ON pf1.PersonID = p1.ID JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID JOIN Person p2
ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

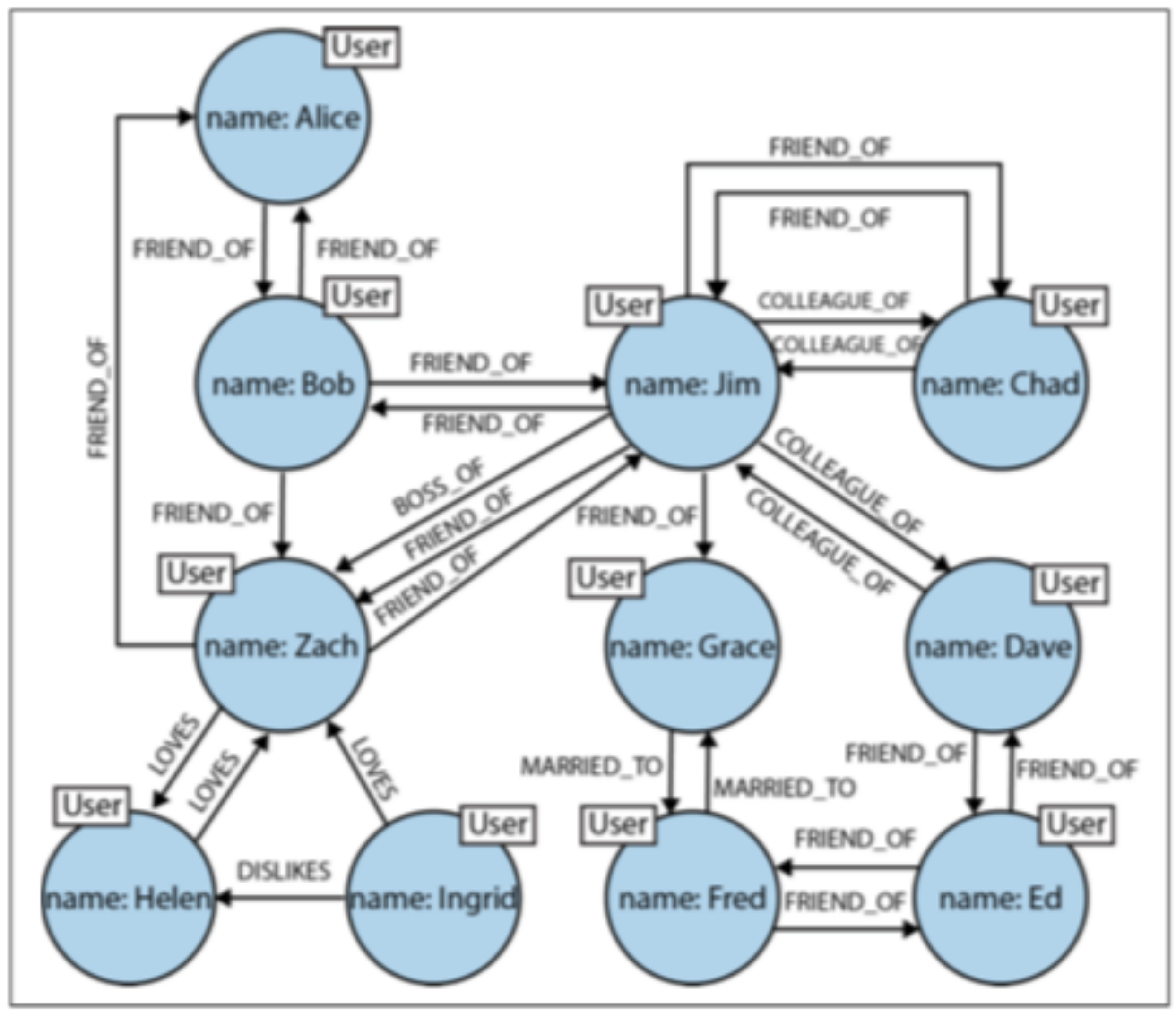
PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Alternatives to Relational Schemes: Graph Models

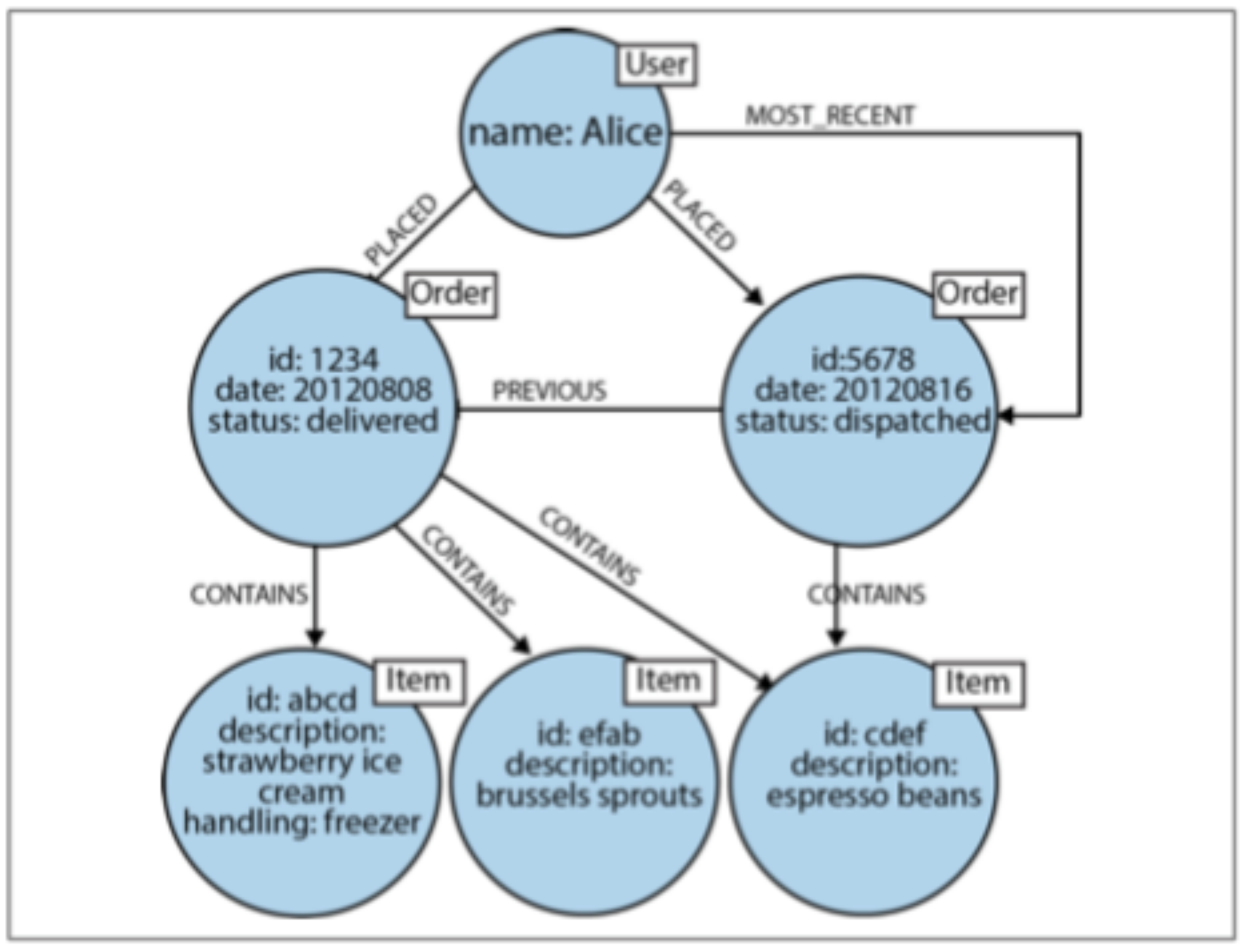
- Property graph model by Neon
  - Each vertex consists of
    - A unique identifier
    - A set of outgoing edges
    - A set of incoming edges
    - A collection of properties — key-value pairs
  - Each edge consists of
    - A unique identifier
    - The tail vertex
    - The head vertex
    - A label to describe the relationship
    - A collection of properties — key-value pairs

Alternatives to Relational Schemes: Graph Models



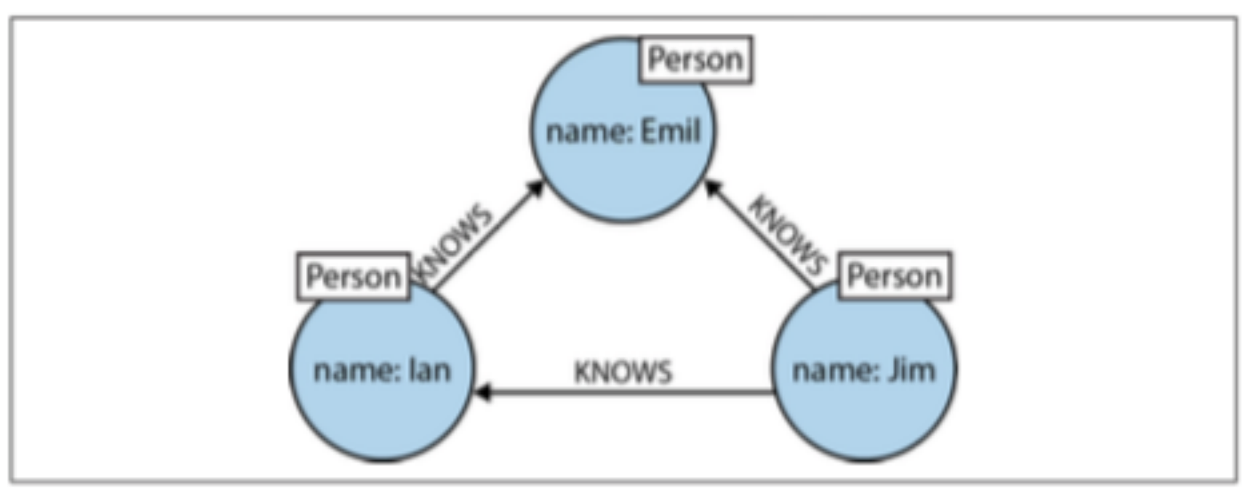
Alternatives to Relational Schemes: Graph Models

- Order history as a property graph



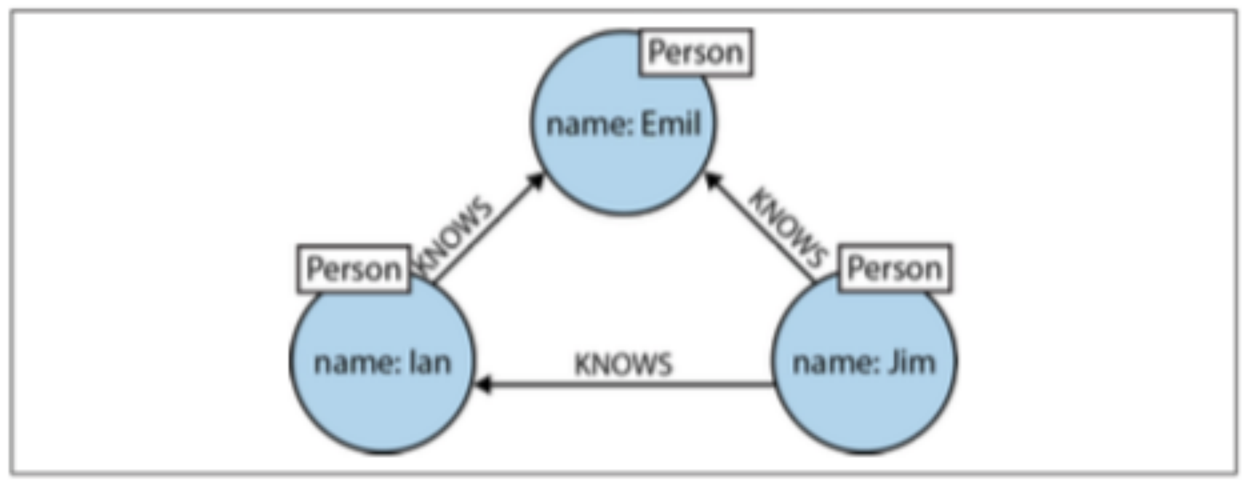
Alternatives to Relational Schemes: Graph Models

- Processing queries in Neo4j
  - Use Cypher (from “The matrix”)
  - Can describe a path




```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

Alternatives to Relational Schemes: Graph Models




```
(emil:Person {name:'Emil'})
      <-[:KNOWS]-(jim:Person {name:'Jim'})
      -[:KNOWS]->(ian:Person {name:'Ian'})
      -[:KNOWS]->(emil)
```

Alternatives to Relational Schemes: Graph Models

- Finding the mutual friends of Jim:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-
[:KNOWS]->(c)
RETURN b, c
```

Alternatives to Relational Schemes: Graph Models

- Triple Stores

- Information is stored as (subject, predicate, object)
  - Subjects correspond to vertices
  - Objects are
    - A value in a primitive data type — (jim : age : **64**)
    - Another vertex — (jim : friend_of : thomas)

Alternatives to Relational Schemes: Graph Models

```
@prefix : </example>
_:lucy    a          :Person
_:lucy    :name      “Lucy”
_:lucy    :born_in   _:idaho
_:idaho   a          :Location
_:idaho   :name      “Idaho”
_:idaho   :type      “State”
_:idaho   :within    _:usa
```

Alternatives to Relational Schemes: Graph Models

- Triple stores are the language of the semantic web
- Semantic web:
  - Machine readable description of type of links
    - e.g. image, text, …
  - Creates web of data — a database of everything
- Stored in Resource Description Framework (RDF)
- SPARQL — query language for triple stores