COMP9321:
Data services engineering

Semester 1, 2018,

Week 6 Accessing and Publishing Data

By Helen Paik, CSE UNSW

We are Generating Vast Amount of Data …

Air Bus A380:

- generate 10 TB every 30 min

Twitter: http://www.internetlivestats.com/twitter-statistics/

- Generate approximately 12 TB of data per day.

Facebook:

- Facebook data grows by over 500 TB daily.

New York Stock:

- Exchange 1TB of data everyday.

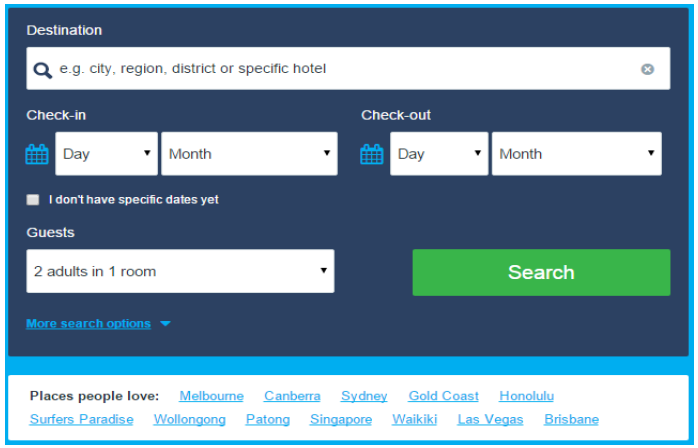
https://www.brandwatch.com/2016/03/96-amazing-social-media-statistics-and-facts-for-2016/

Challenge

How do we store and access this data ?

E-Commerce website

- Data operations are mainly transactions (Reads and “Writes”)
- Operations are mostly on-line
- Response time should be quick but important to maintain **security** and **reliability** of the *transactions.*
- ACID properties are important

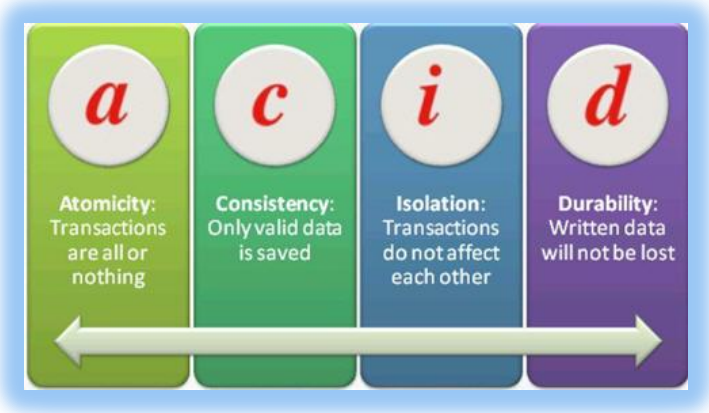


Challenge

How do we store and access this data ?

Image serving website (many social network sites in general)

- Data operations are mainly fetching information (Reads)
  - also the “fan-out” effect is challenge
- Operations are mainly on-line
- High bandwidth requirement
- ACID requirements can be relaxed



Challenge

Fan-out …

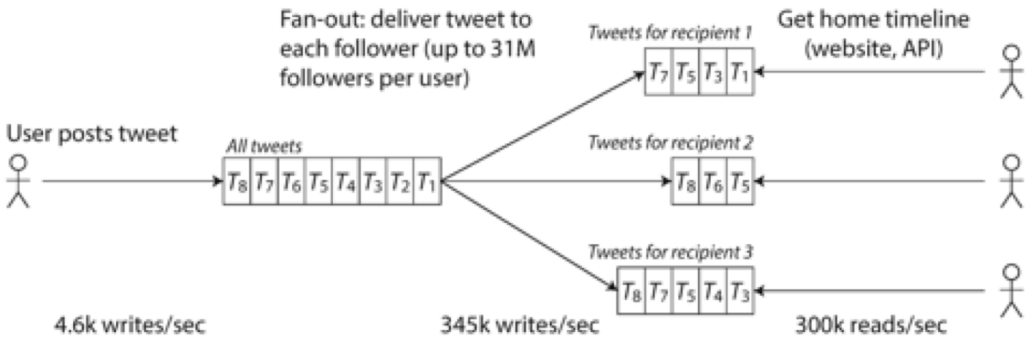


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

A user can see tweets posted by the people they follow …

- A new post –> look up the followers and 'write' to each follower's timeline ahead of time -> makes reading easy
- But this also creates a lot of 'writing' work
  - On average 75 followers, but can vary widely (some users have 30 million followers)
  - May need to consider the distribution of the followers per user (and how often each user tweets)

Challenge

- How do we store and access this data ?

Search Website

- Data operations are mainly reading index files for answering queries (Reads)
- Index compilation is performed off-line due to the large size of source data (the entire Web)
- ACID requirements can be relaxed
- Response times must be as fast as possible.

Challenge for API …

How do we store and access this data over the web ?

- Consumption of Data (for you to take data in …)
- Publication of Data (for you to make data available for others …)

Important question: What is your data model behind the API?

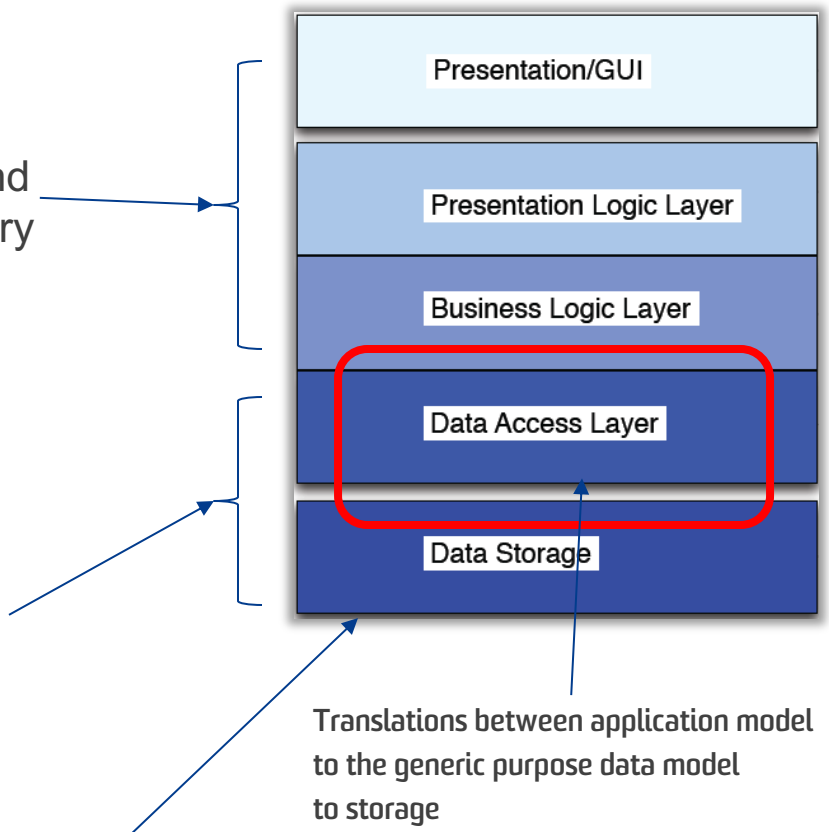
Data models can change how we think about the problem you are solving

What is in a data model …

An application developer “thinks” in terms of the real world (people, organisations, actions, goods, etc.) … and model it as objects/data structures and APIs that manipulate them – these models are very specific to each application

When you want to store the objects, you express them in generic-purpose data model such as JSON, XML documents or tables.

The “storage” also allows the representation to be queried, searched or manipulated.



The engineers of the ‘storage solution’ software decide on how JSON/XML/tables are represented in terms of bytes in memory, disk or on a network.

Relational Model vs. “NoSQL” Models

Relational Model (more or so synonymous with SQL)

- The best known, probably the most successful data model which has proven itself in many aspects to be the data model of choice in many applications
- Data is organised into relations (table) where each relation holds an unordered collection of tuples (rows)

Based on solid theory and well engineered implementation -> many competing models have been proposed, but never managed to take over SQL

Built for business data processing

- Typical business transactions (airline reservations, stock keeping, etc.)
- Batch processing (invoicing, payroll, reporting, etc.)

Turned out it was still generically applicable to many modern Web applications too

Hypothetical Relational Database Model

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Stree, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sep-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1852	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution

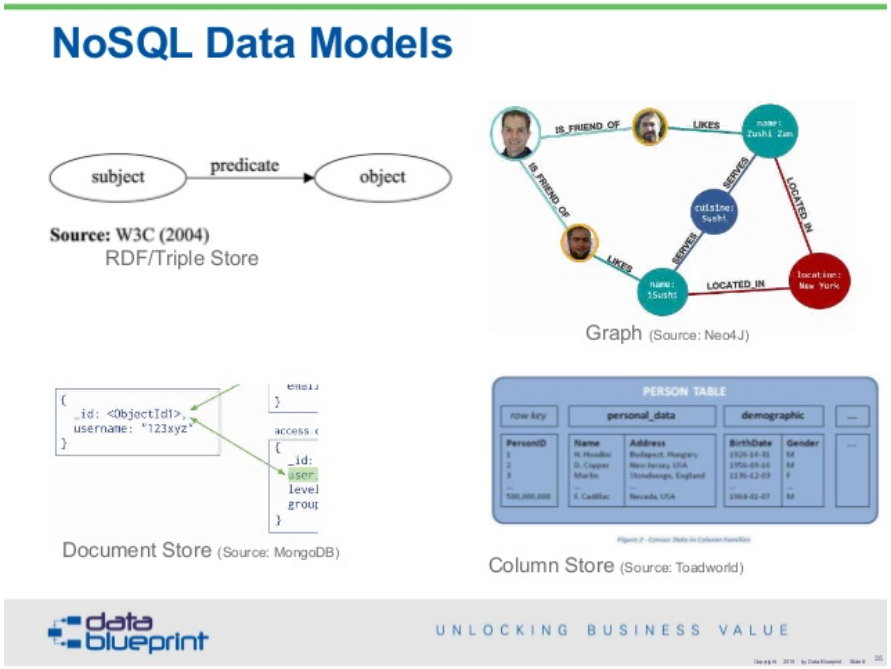
Relational Model vs. “NoSQL” Models

The rise of NoSQL … (since 2010 or so)

- Refers to a host of technologies that implement distributed, “non-relational” databases

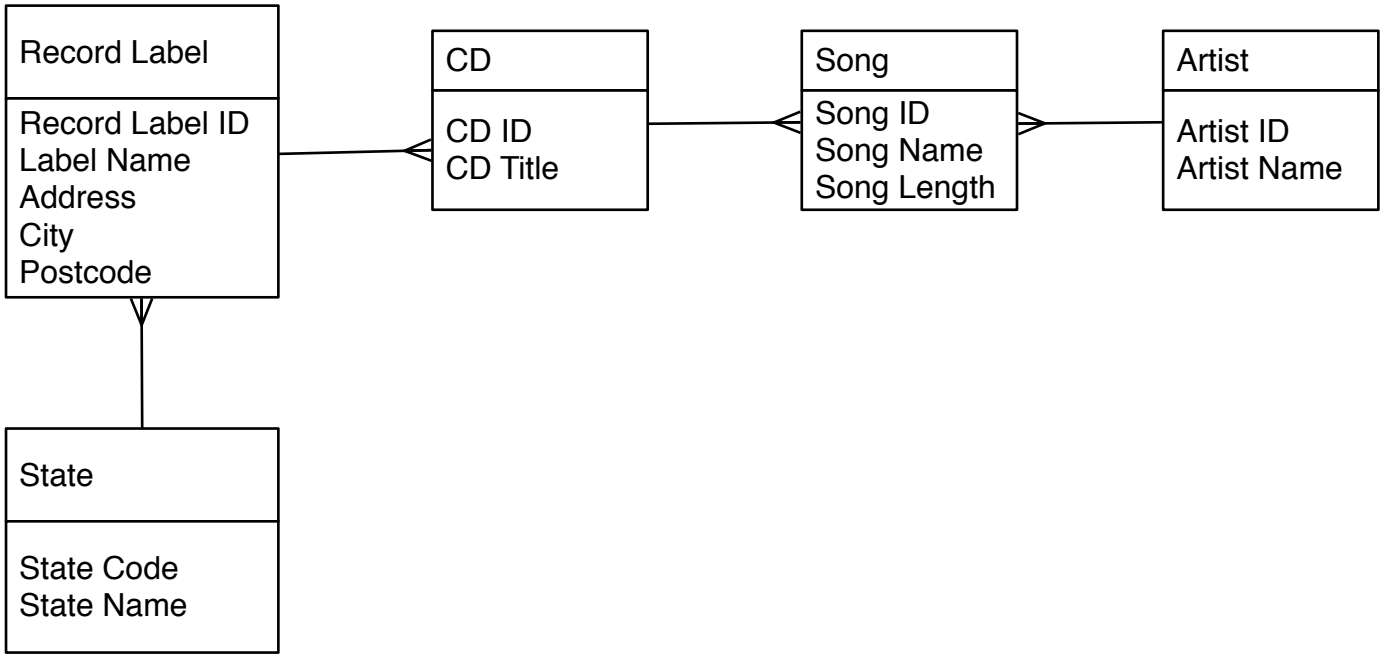
Why NoSQL?

- A need for greater scalability – very large datasets or very high ‘write’ throughput
- A need for more expressive and dynamic data model
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties



Designing Data-Intensive Applications, by Martin Kleppmann
Image: https://www.slideshare.net/Dataversity/trends-in-data-modeling

Problems with Relational Models



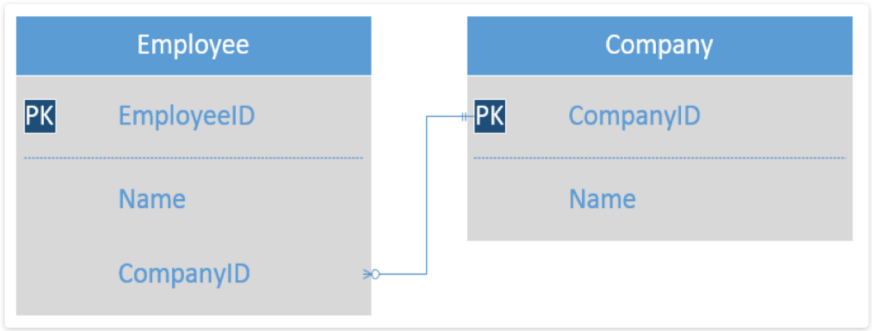
Normalisation … 3NF

* many fragments -> leading to many joins -> scalability ?

Problems with Relational Models

The Object-Relational Mismatch (Impedance Mismatch)

- Refers to the problem of a mismatch between application data model (your business objects) and data model for storage (in relational tables)
- This mismatch creates a need for an awkward translation layer between the objects in the application code and the database model of tables/row/columns.



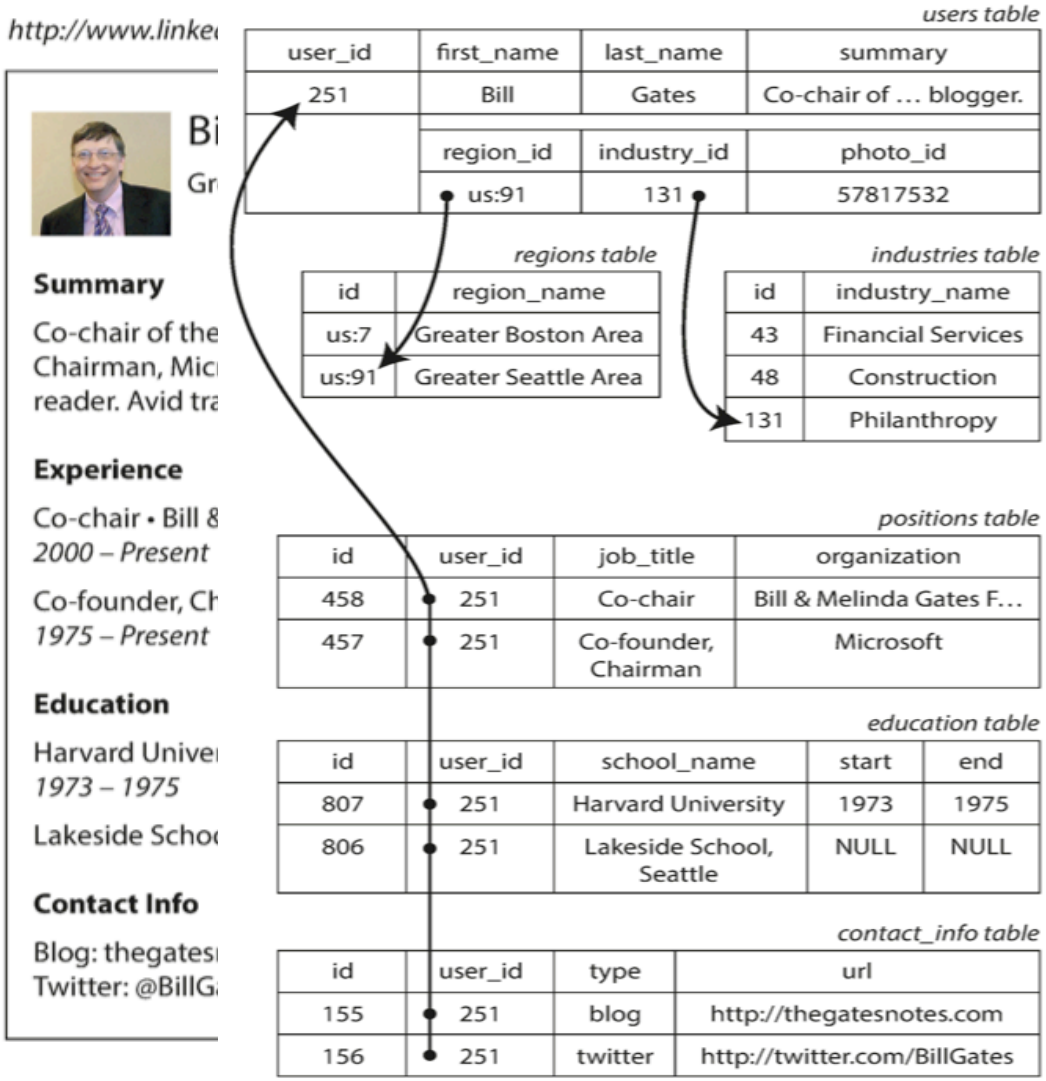
Employee and Company

```
public class Employee
{
    public string Name { get; private set; }
    public Company Company { get; private set; }
}

public class Company
{
    public string Name { get; private set; }
    public List<Employee> Employees { get; private set; }
}
```

Alternative Data Models?

Relational Modelling
of a resume (e.g., LinkedIn Profile)

Typical normalised form
Would put multi-values in
separate tables with user_id
as foreign key

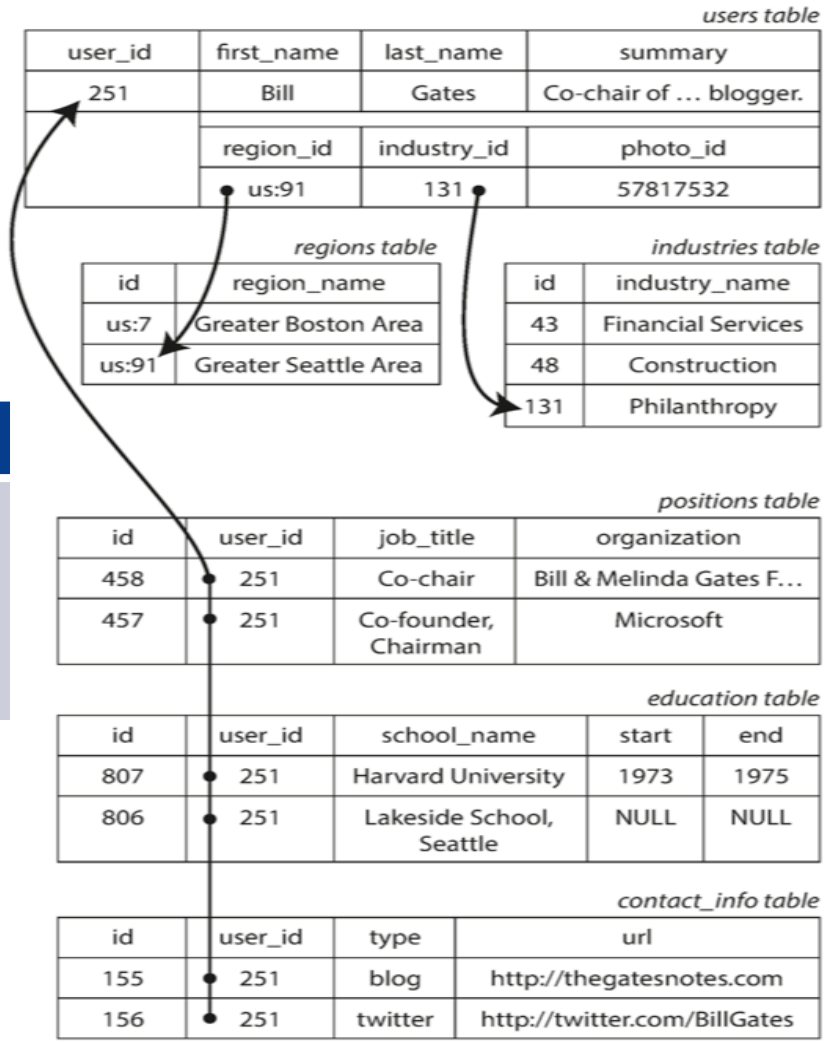


Added features in SQL …

Some databases support an idea similar to 'Arrays':

- an store multi values in a single row
- can be queried and indexed

User_id	…	Job_title	School_name
251		{co-chair, Bill & Melinda Gates ..}, {Chairman, Microsoft)	{Havard University, 1973,1975},{Lakeside School, Null, Null}



Alternative Data Models?

Example 2-1. Representing a LinkedIn profile as a JSON document

```
{
  "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"
  }
}
```

Another option:

- Encodes jobs, education, contact info as a JSON (or XML) document
- Stores the whole document in a text column in the database
- Application code accessing this info will have to deal with the structure as a whole
- You cannot use the database to query for values inside the column

Document-based databases support this idea naturally (e.g., MongoDB – insert/query JSON objects)

Document-based databases

MongoDB (the most well-known example)

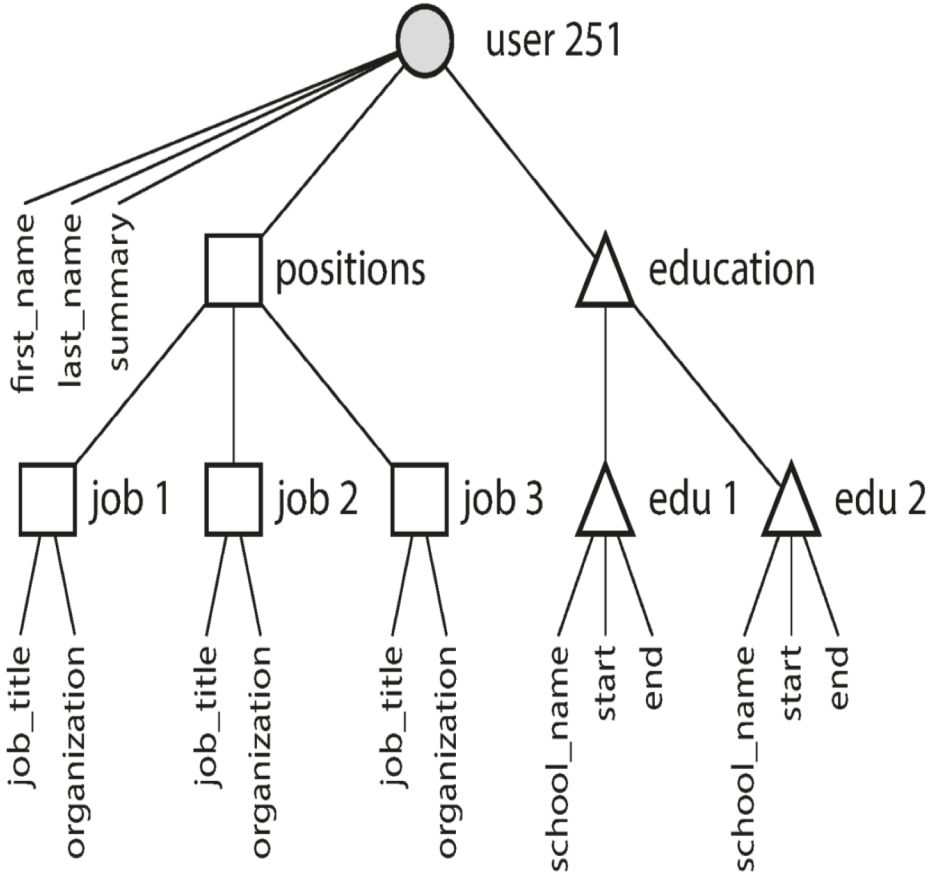
RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

Notable points:

- Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose
- No joins (everything embedded in a single object)

https://www.tutorialspoint.com/mongodb/mongodb_overview.htm

Document-based databases



Embedded objects normally are the result of One-to-Many relationships

Improved “locality”

- a single retrieval request is enough to get all necessary into on “User”

The mismatch between application data model and storage-purpose data model is significantly reduced

- “Create a User” (JSON) in app code and “Insert a User” (JSON) into Document Collections

Document model is not good with …

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

What about Many-to-One or Many-to-Many?

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.
	region_id	industry_id	photo_id
	us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

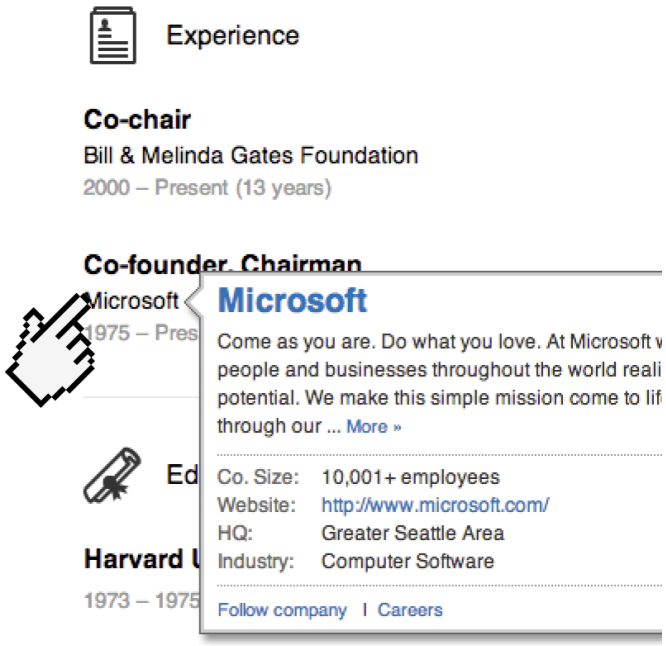
The relational model based solution of these “look-up tables” are useful:

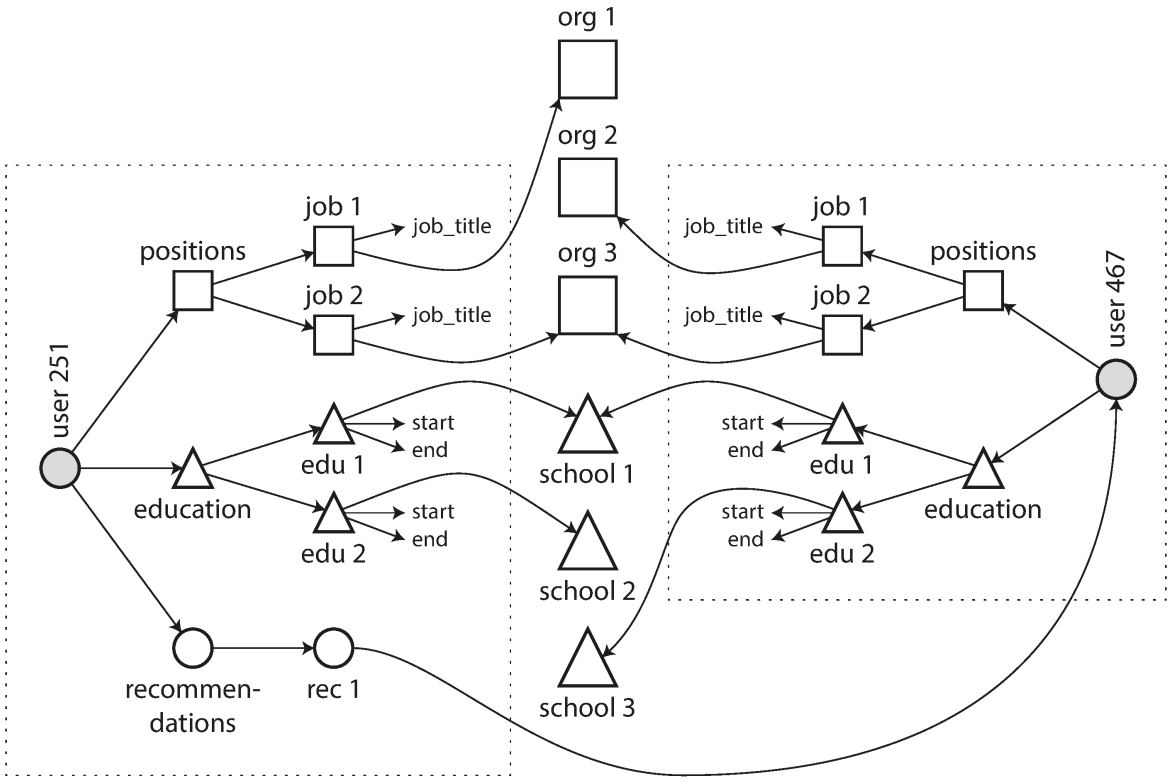
- Consistent style and spelling across Users
- Avoiding ambiguity (e.g., if several cities with the same name)
- Ease of updating (the name is stored in only one place)
- Better search – a search for philanthropists in the state of Washington can match this User 251 (via another table)

Storing ID vs Text -> NOT duplicating text is more flexible and keeps data consistent – reason for normalising in RDB

Document model is not good with …

The single “documents” tend to be come more interconnected as more features are added





The company – linking it as a full entity by itself
(Many-to-One Relationship)

The recommendations – linking it to other Users
(Many-to-Many Relationships)

Relational vs. Document

When it comes to representing many-to-one and many-to-many relationships, both are not that different …

- Foreign keys (ID references) in relational
- Document references (Doc ID) in document-based

The IDs are resolved at retrieval time by using a join or follow-up queries.

- But joins on M-M or M-1 relationship are a routine – highly optimised at the database level
- Document models – join support could be weak, application code might have to resolve the relationships as needed

Relational vs. Document

Which data model leads to simpler application code?

- If the application data model looks like a tree (document-like) -> it can be loaded at once using document-based model
- If M-M relationships are central to the application data model -> relational model is efficient in joins. If document model is used, some of the 'join' logic will have to move to application code

Consider the kinds of relationships between data items. If they are highly interconnected data (e.g., social network)

- document model is not so good,
- relational model is OK …
- graph models would be natural (to be seen later)

Relational vs. Document

Schema flexibility, always a good thing?

- Most document-based databases do not enforce any schema in documents (schema-less databases)
  - Arbitrary keys and values can be added to a document and when reading clients have no guarantees as to what fields the documents may contain
- Schema-on-read
  - The structure of the data is implicit, only interpreted when the data is read by application code
  - ≈ dynamic (runtime) type checking
- Schema-on-write
  - The traditional approach of RDB - explicit schema and the database ensures all written data conforms to it
  - ≈ static (compile-time) type checking

Relational vs. Document

Schema flexibility, always a good thing?

- When does this ‘schema-on-read/write’ matter? -> when application wants to change the format of its data.
- E.g., User name in one field -> User name in two fields.

(Document Based)

```
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];
}
```

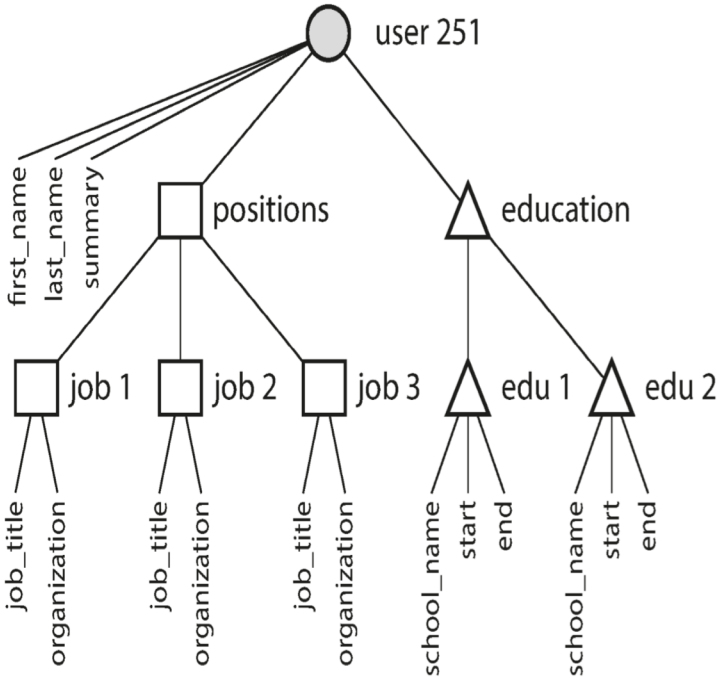
(Relational DB)

```
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
```

DOC model is considered advantageous if the docs in the collection tend to have different structures (e.g., different types of related objects)

Relational vs. Document

Data locality for queries - doc-based systems store a document as a single continuous string as JSON or XML (or a binary variant)



users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.
	region_id	industry_id	photo_id
	us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

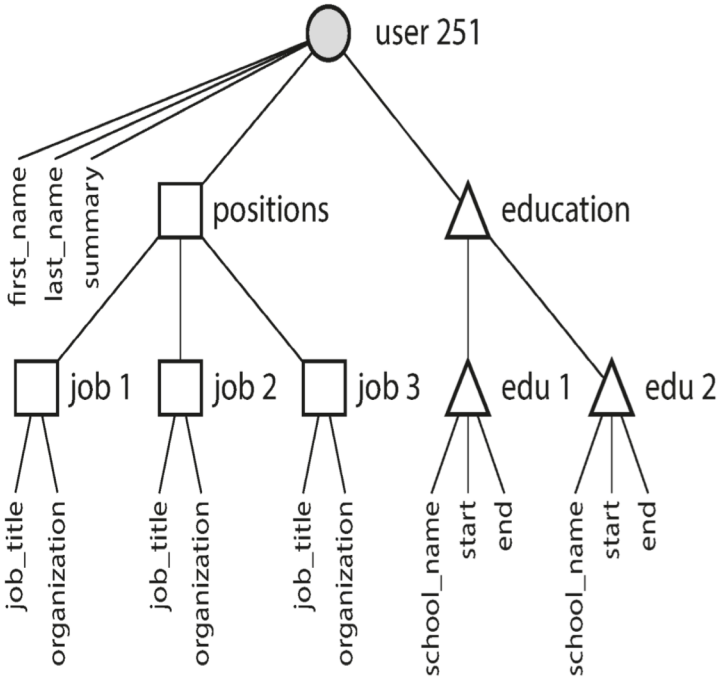
contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

If your application requires the entire document (e.g., to render it on a Web page as a whole), there is a performance advantage over split tables

Relational vs. Document

The locality of Doc-based systems

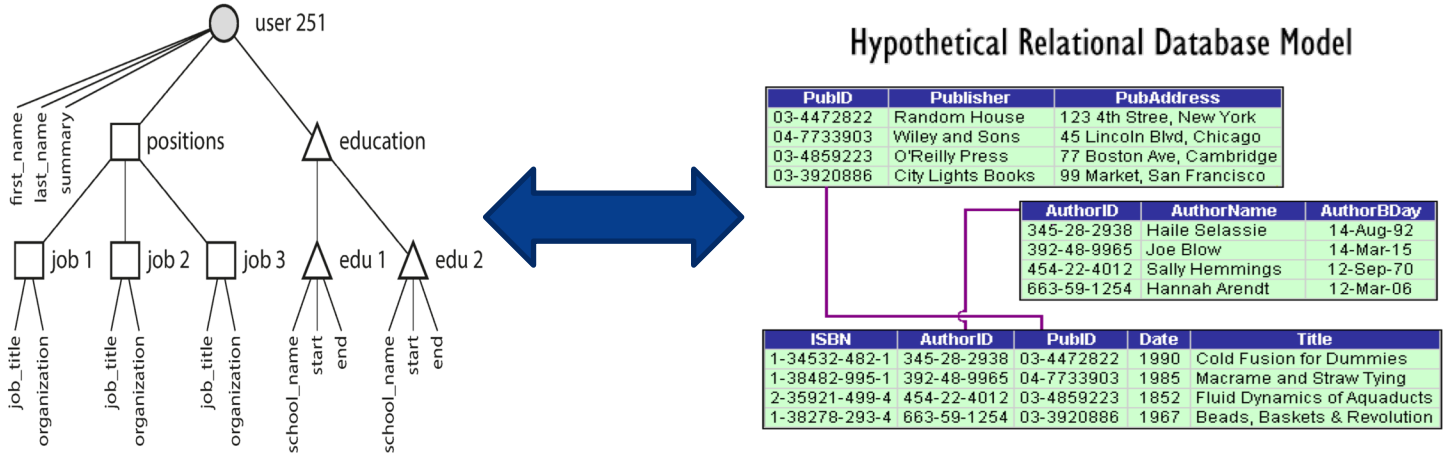


Data locality advantage only applies only if you need large parts of the document at a time (often the whole document needs to be loaded only if you need to access a small portion of it)

On Updates, normally the whole document needs to be rewritten (except tiny changes that do not change the overall encoded size of the document)

Relational vs. Document

Convergence of document and relational databases



PostgreSQL (since v.9.3), MySQL (since v.5.7). IBM DB2 (since v.10) support JSON documents.

RethinkDB, MongoDB (document-based) support relational-like joins in its query language

The two models can complement each other -> A hybrid model seems like a trend in these two systems

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;
}
```

Most programming languages are imperative:

- step-by-step instructions on how the data should be returned …

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

Most query languages are imperative:

- Specify the pattern of data to be returned, not how it is returned
- The database is optimised on how to do this

This declarative query paradigm is the same in Relational or Document-based systems

Query languages for data

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

Most query languages are imperative:

- Specify the pattern of data to be returned, not how it is returned
- The database is optimised on how to do this

Declarative query paradigm is good for many reasons:

It is up to the database to decide which index tables to consult, which joins should be performed, in which order the various parts of the query are executed

The query language can remain concise

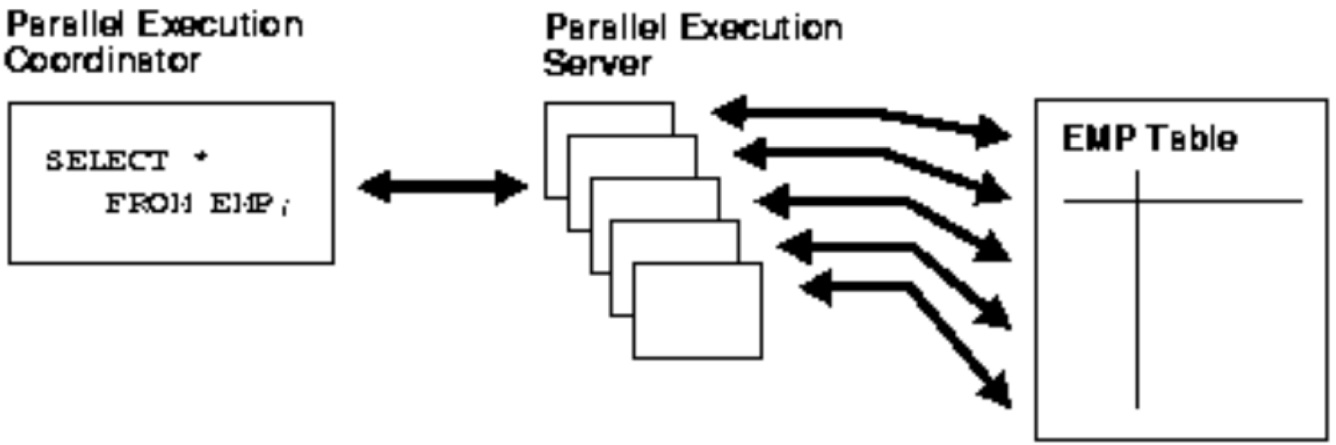
Any further optimisation/performance improvement of the database system can happen without affecting the query interface (e.g., re-arranging disk space)

Suitable for parallel execution strategy …

Query languages for data

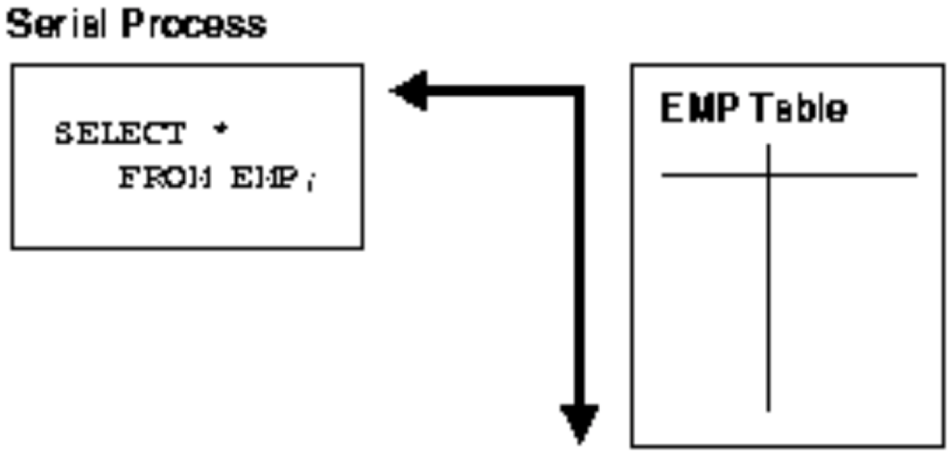
Most query languages are imperative:

```
SELECT * FROM ani
```



Declarative query p

It is up to the datab
performed, in w



f the database system can happen
nging disk space)

Query Languages for Data

Accessing DB from an Application:

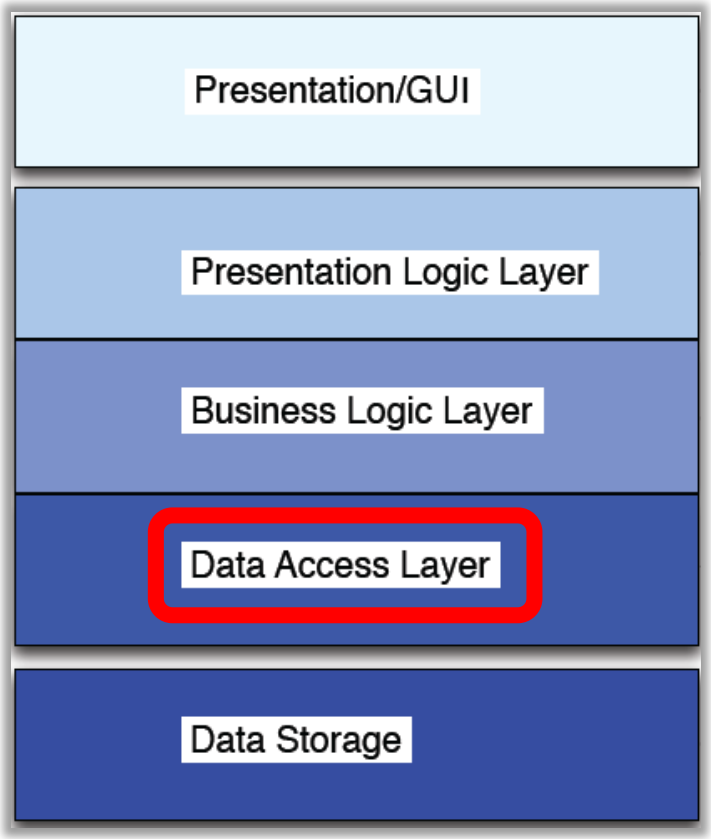
When you work with a database system (regardless of its storage model) in an application, the code issues a query statements to the database via some form of “data-connectivity API”

Database connectivity API specifications

- Java has JDBC API, Python has DB-API, Microsoft variety has ODBC API, etc.

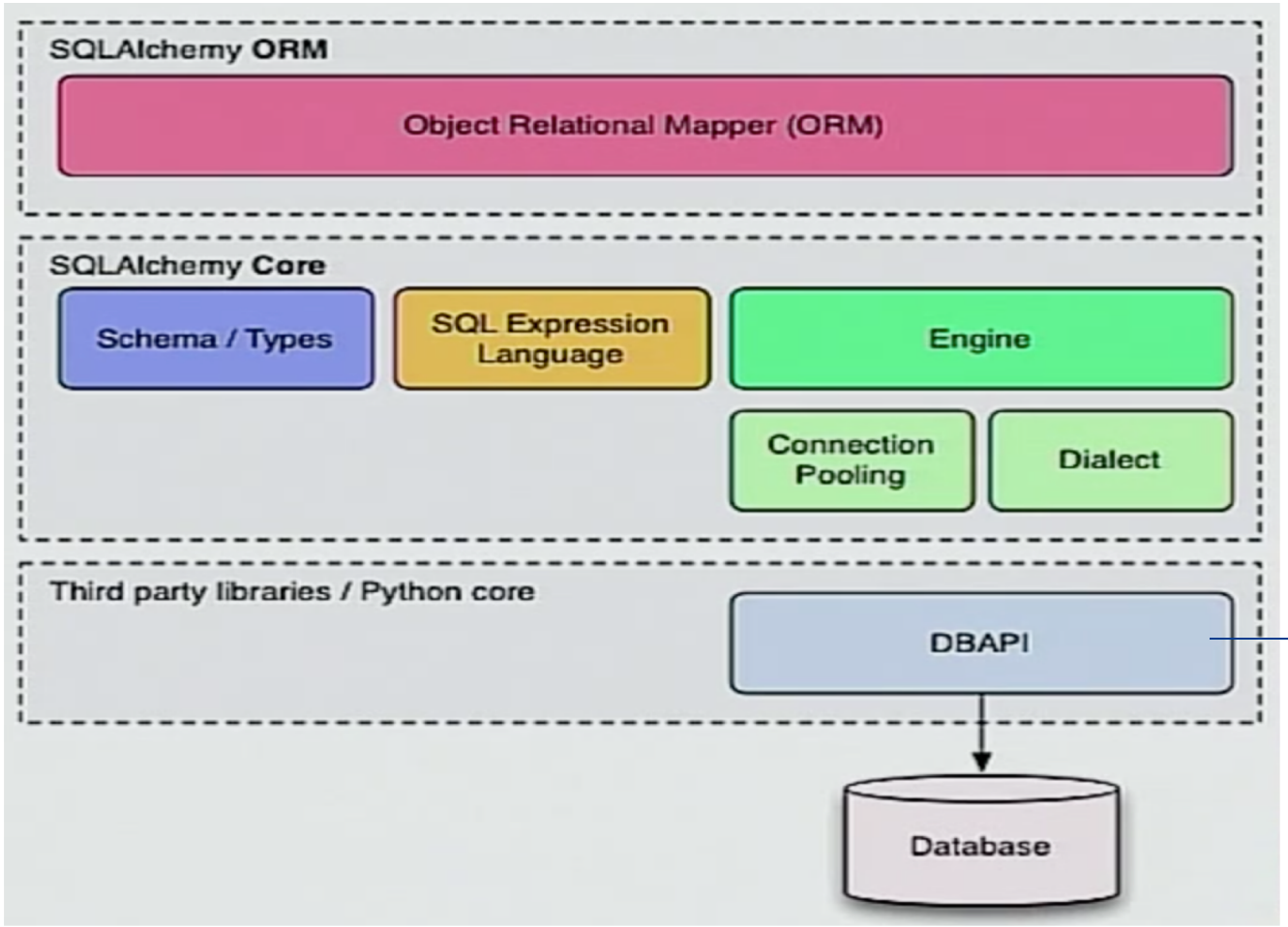
Each specification is then implemented by the database system provider as a library for the developers (e.g., DB-API library for PostgreSQL, or JDBC library for Oracle)

The application code blocks relating to using this library form “Data Access Layer” in the stack.



Directly Executing SQL

e.g., Python SQLAlchemy



(Optional)

SQL access management libraries"

Python standard/core Low-level database communication layer (many implementations)

- psycopg2 (PostgreSQL)
- cx_oracle (Oracle)
- pyodbc (ODBC)

Directly Executing SQL

DBAPI – e.g., psycopg2

```
import psycopg2
connection = psycopg2.connect("scott", "tiger", "test")

cursor = connection.cursor()
cursor.execute(
          "select emp_id, emp_name from employee "
          "where emp_id=%(emp_id)s",
          {'emp_id':5})
emp_name = cursor.fetchone()[1]
cursor.close()

cursor = connection.cursor()
cursor.execute(
          "insert into employee_of_month "
          "(emp_name) values (%(emp_name)s)",
          {"emp_name":emp_name})
cursor.close()

connection.commit()
```

Connect
(network/or file handle)

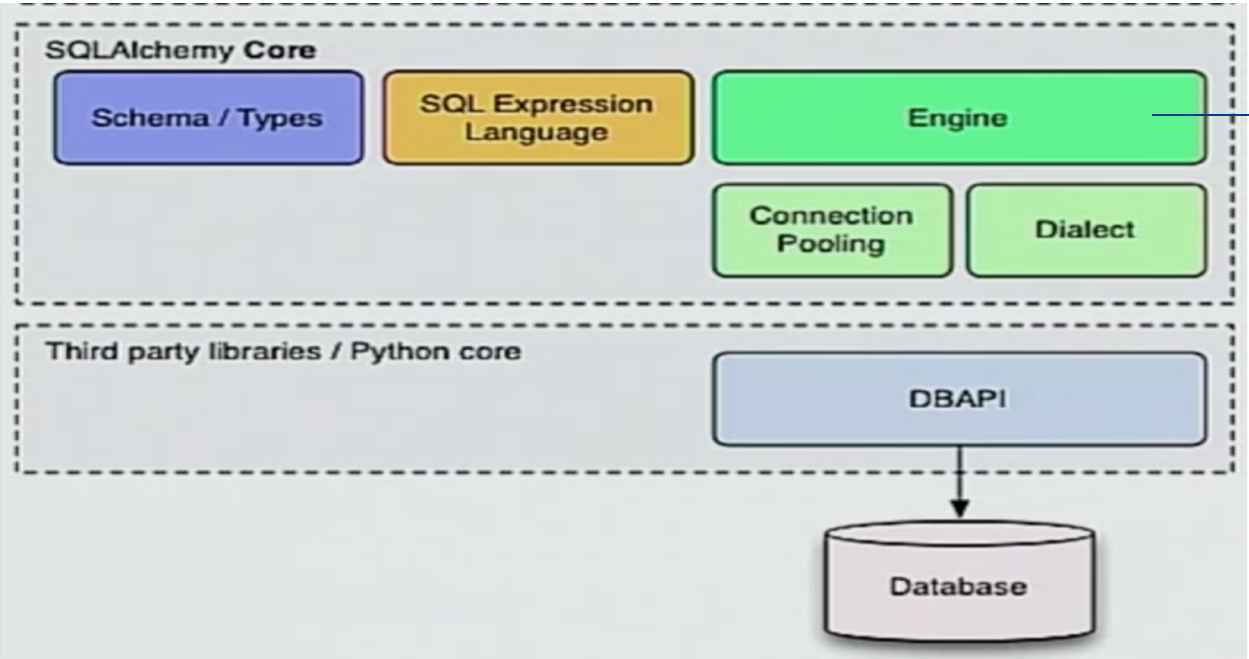
"An object for
Table rows and loops
within them"

Bound parameter

Directly Executing SQL

DBAPI

- Many different implementations of the spec …
- Inconsistency between different implementations (e.g., bound parameter formats, exception hierarchy)
- No explicit transaction markers (no begin() transaction)



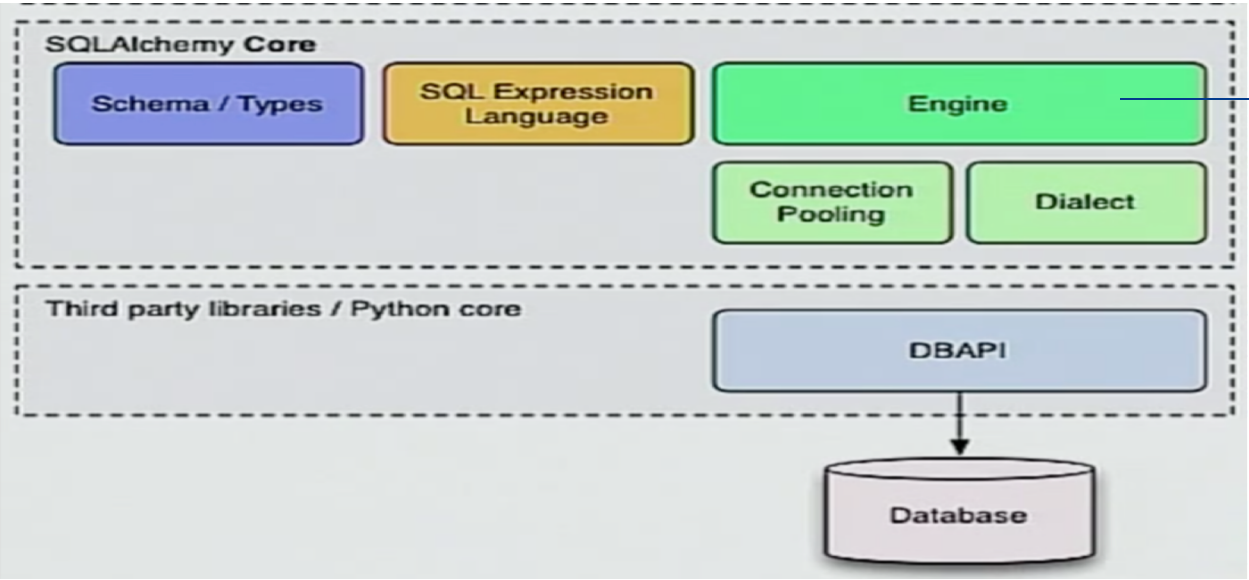
Object that maintains
The classical DBAPI interactions
With the database

Directly Executing SQL

SQLAlchemy

- “Uniform” SQL access to relational databases (in SQLAlchemy way)
- i.e., SQL access library built on top of the DBAPI connectivity

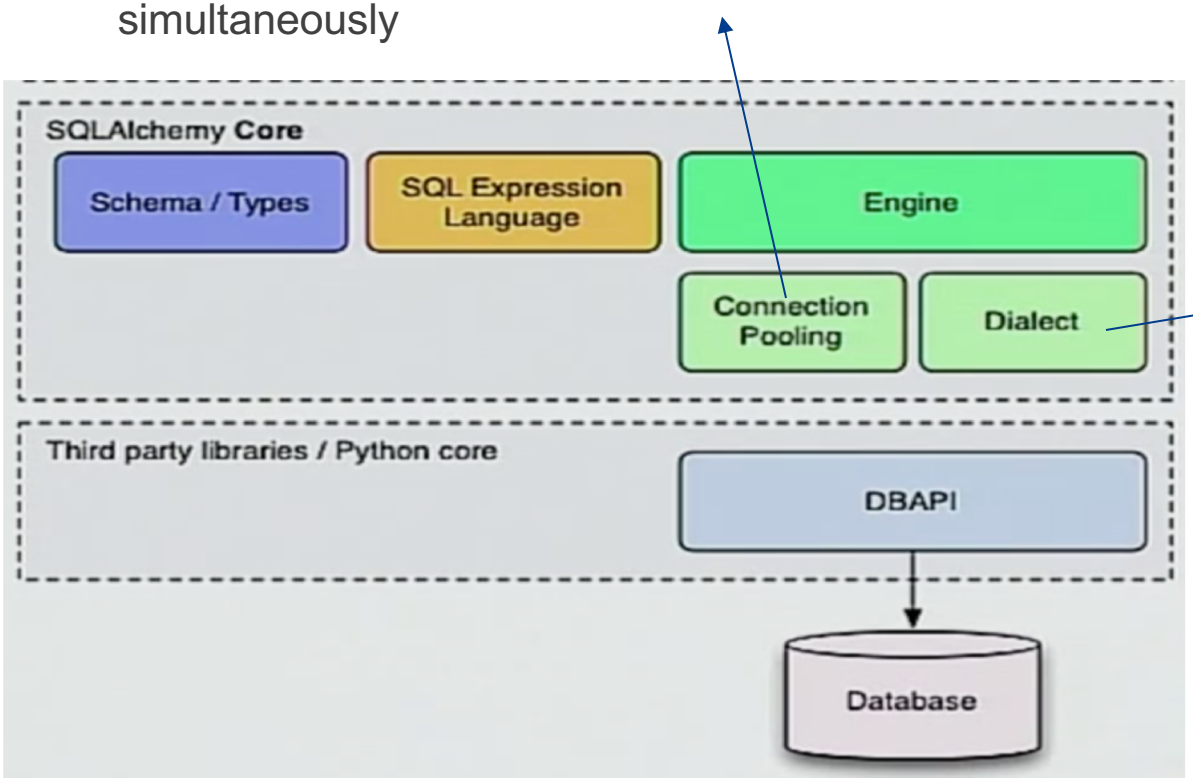
```
from sqlalchemy import create_engine
engine = create_engine('postgresql://usr:pass@localhost:5432/sqlalchemy’)

create_engine engine = create_engine(‘sqlite:///some.db’)
```



Object that maintains
The classical DBAPI interactions
With the database

Directly Executing SQL

- Connection Pooling (≈ connection sharing)
  - Creating DB connections are expensive
  - With pooling, program fetches an existing connection, use and put it back into pool
  - easier management of the number of connections that an application might use simultaneously



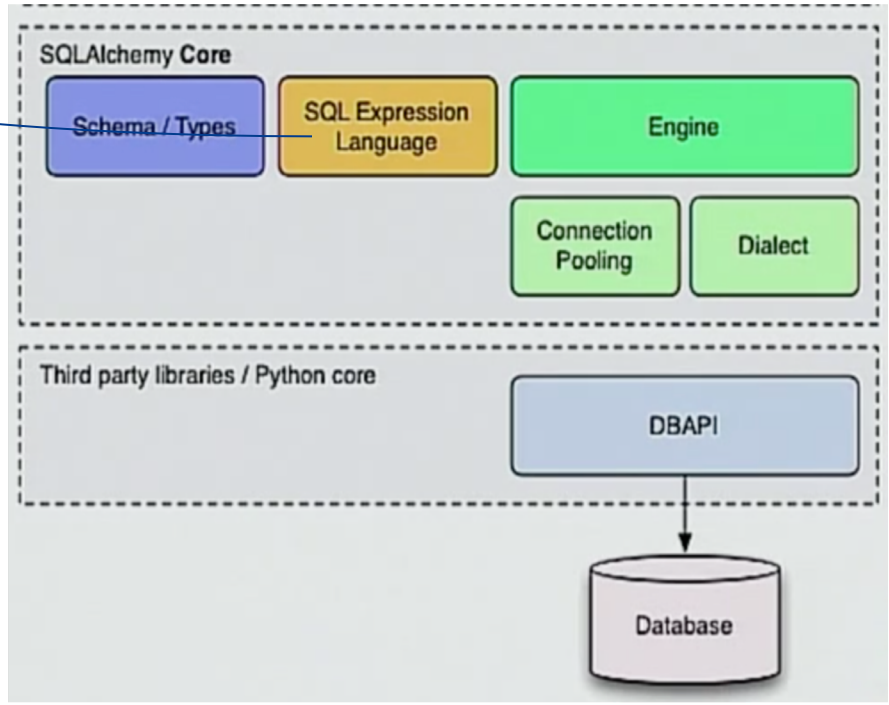
SELECT TOP 10 * FROM people;

Or

SELECT * FROM people LIMIT 10;

Directly Executing SQL

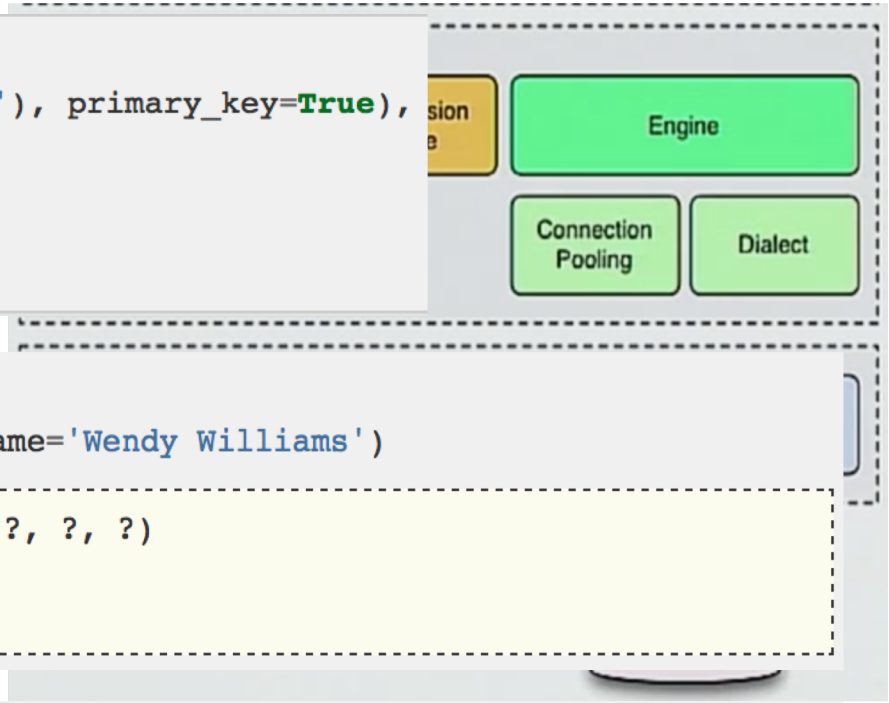
SQL Expression Language



http://docs.sqlalchemy.org/en/latest/core/tutorial.html

Directly Executing SQL

```
users = Table('users', metadata,
    Column('id', Integer, Sequence('user_id_seq'), primary_key=True),
    Column('name', String(50)),
    Column('fullname', String(50)),
    Column('password', String(12))
)
```




```
>>> ins = users.insert()
>>> conn.execute(ins, id=2, name='wendy', fullname='Wendy Williams')
```

```
INSERT INTO users (id, name, fullname) VALUES (?, ?, ?)
(2, 'wendy', 'Wendy Williams')
COMMIT
```

```
>>> from sqlalchemy.sql import select
>>> s = select([users])
>>> result = conn.execute(s)
```

```
SELECT users.id, users.name, users.fullname
FROM users
()
```

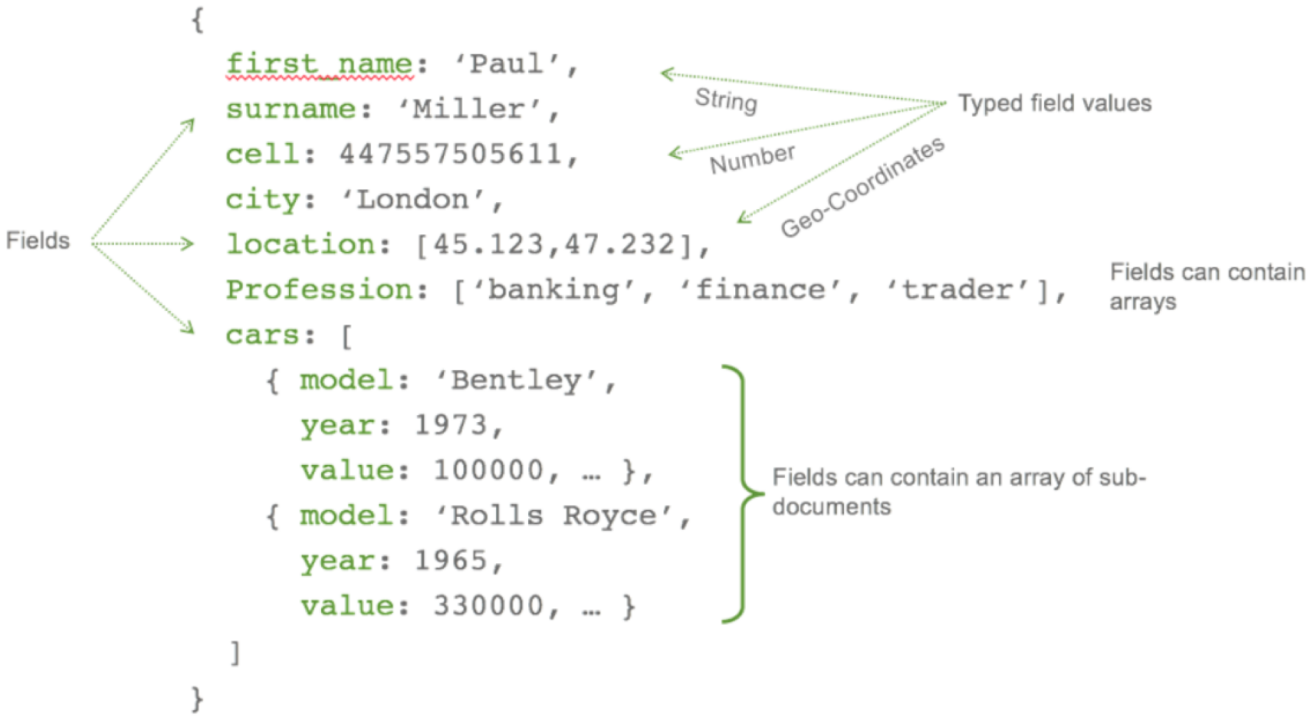

http://docs.sqlalchemy.org/en/latest/core/tutorial.html

MongoDB Query

The basic idea of databased connectivity API applies with MongoDB too …

Many implementations

- Direct: PyMongo, Motor
- ORM-like: PyMODM, MongoEngine, etc.

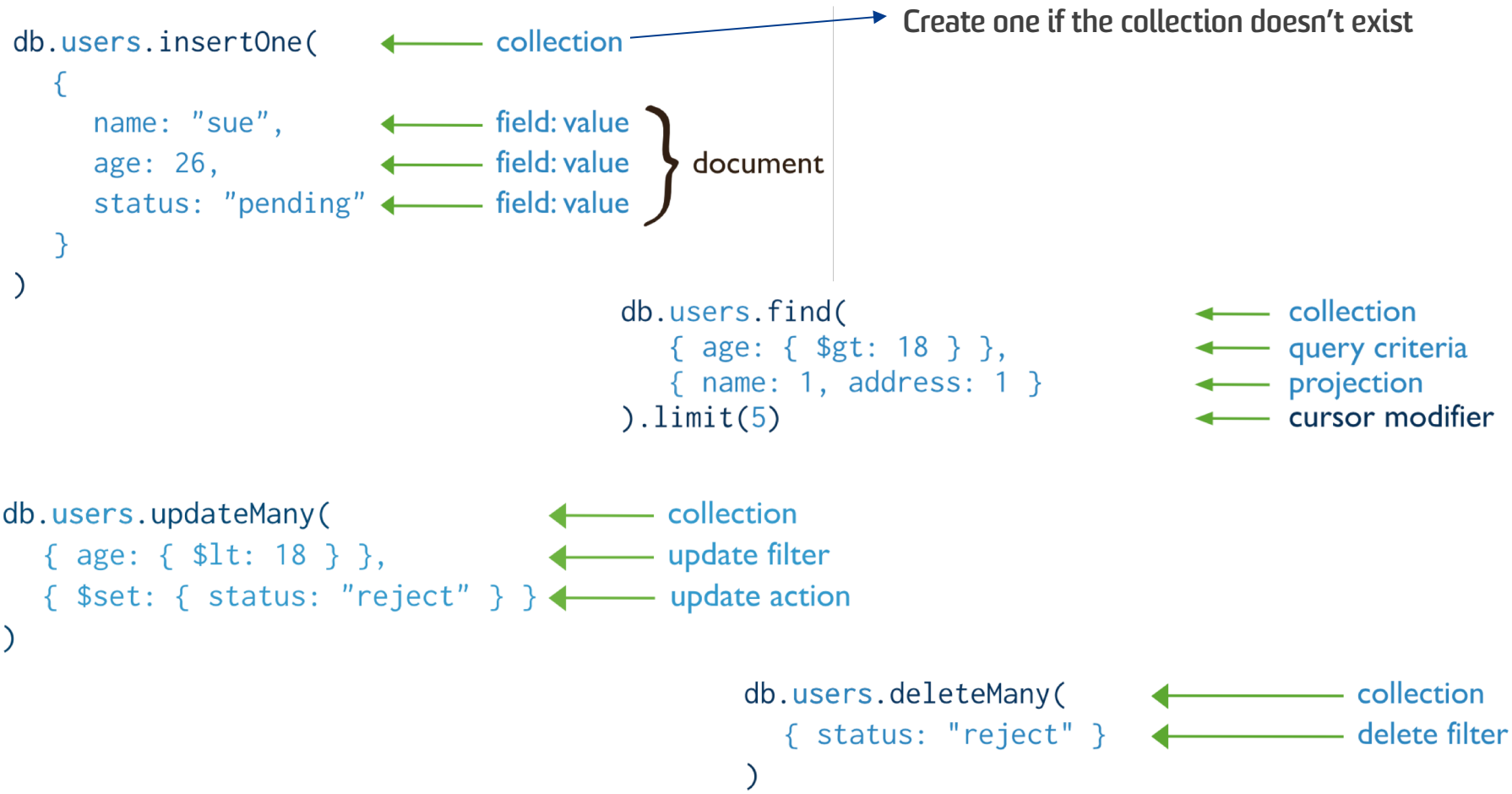


JSON Documents as the first class citizens

https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb

MongoDB Query

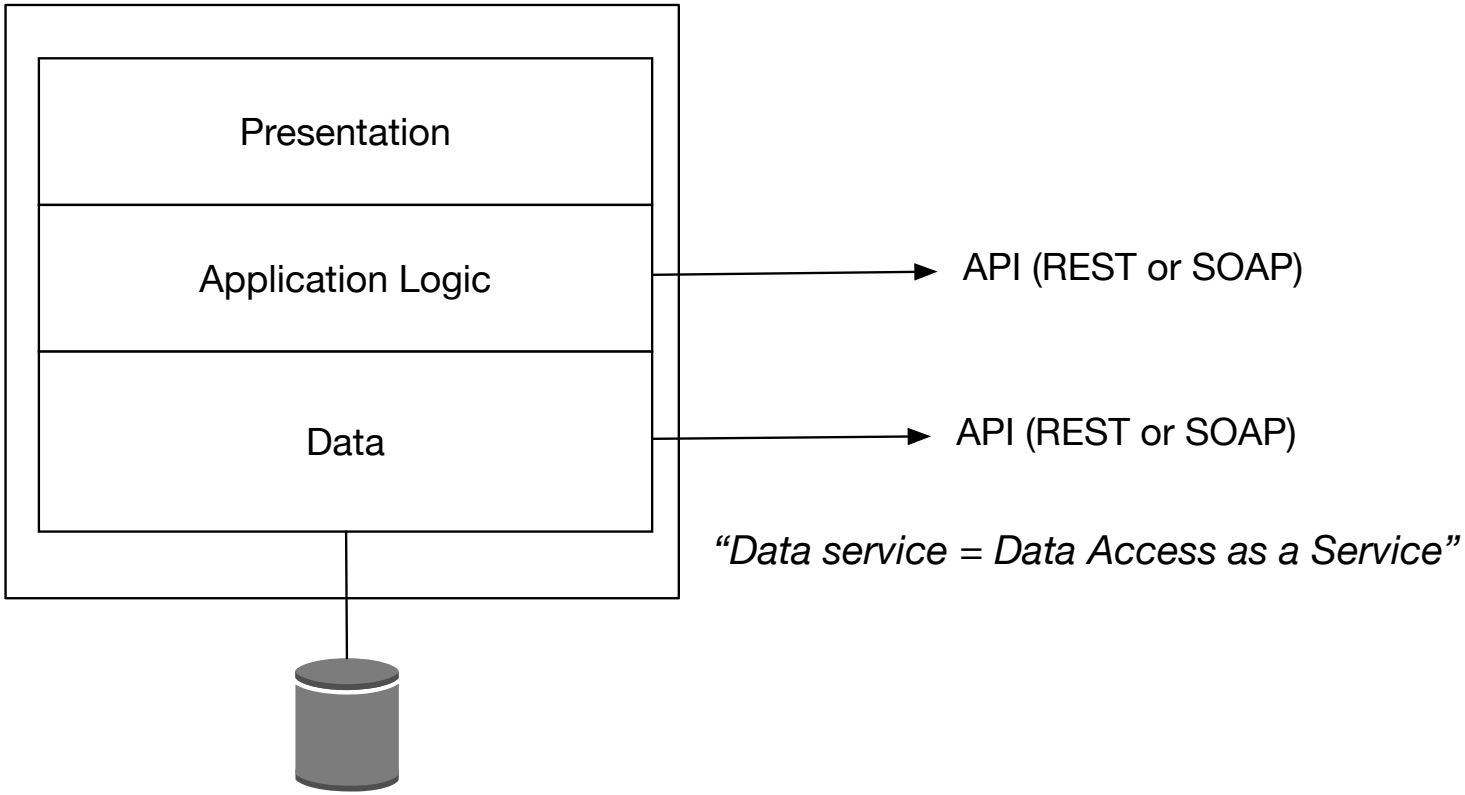
MongoDB Server (download, install, run …) and MongoDB Client (connect, create db, …)



https://docs.mongodb.com/manual/crud/

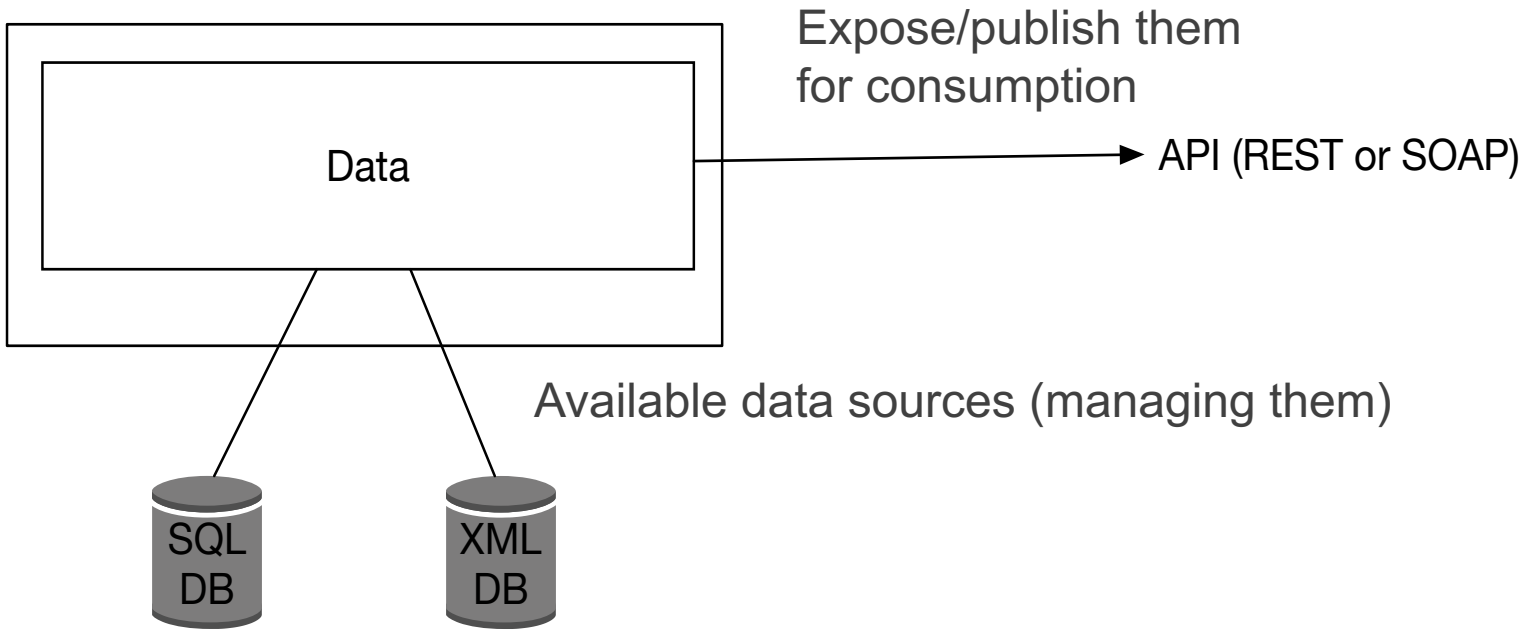
Data Services: accessing/publishing data

Simplistic View



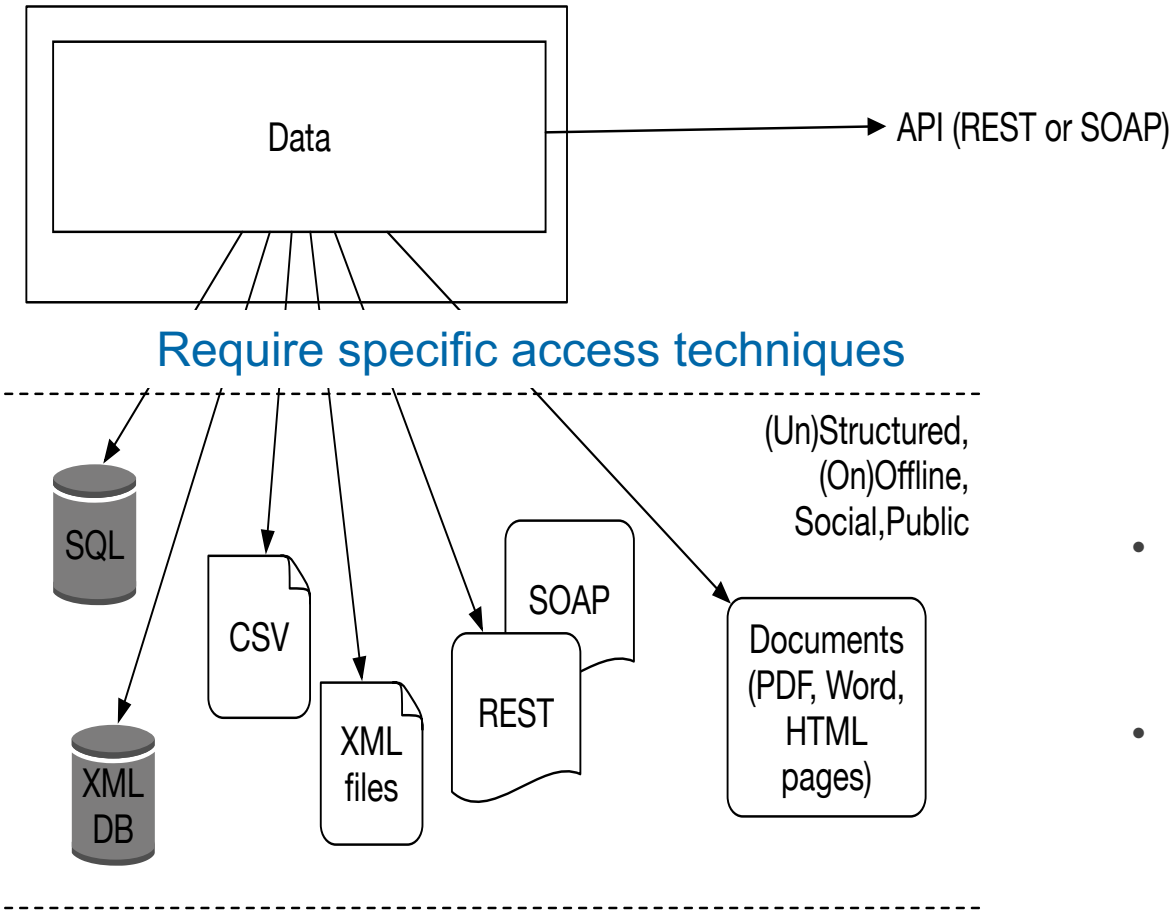
Data Services – what is it about?

Let's consider "Data Layer" only …



Data Services – what is it about?

Two sides of a coin:



- Data integration/aggregation from multiple sources (data prep)
- Data publication for consumer access (API)

Challenges from implementation view point

Difficult to obtain a ”single view of X” for any X

- What data do I have about X?
- How do I stitch together the info I need?
- What else is X related to?

No uniformity (model or language)

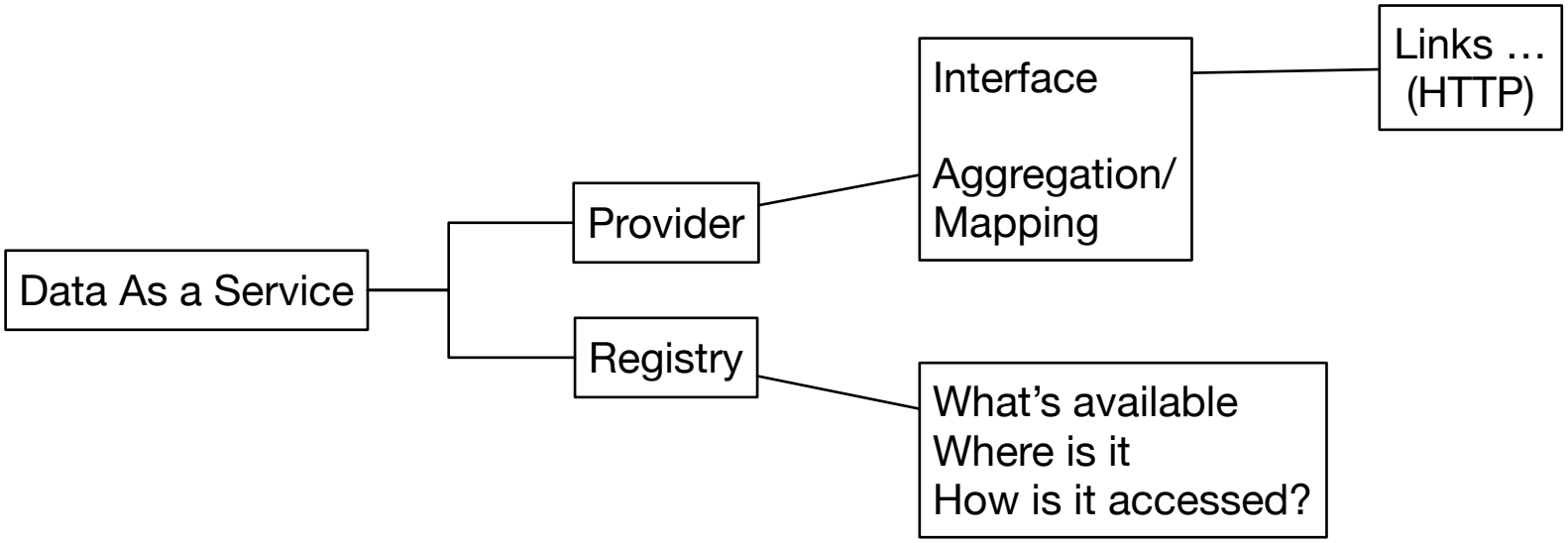
- Data about X is stored in many different formats.
- Accessing or updating X involves many different APIs or access methods
- Manual coding of ”distributed query plans”

What’s data sources or existing APIs available and where?

What protocol do they use?

What format are they in?

DATA as a service should include:



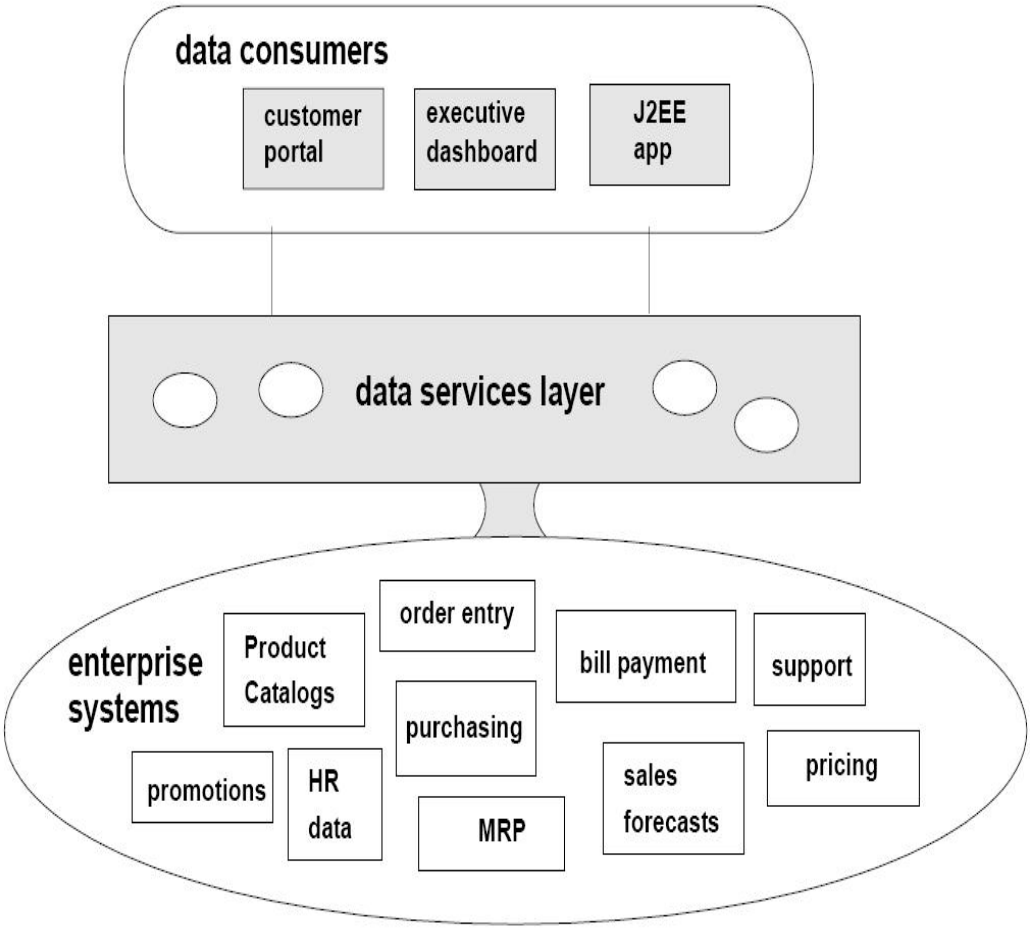
Necessary to have
platform neutral, language independent
data aggregation and publication methods

Data as a service

Sourcing, processing and delivery of data in a usable format

Often the idea includes designing and implementing a master/central hub (authoritative source) for the data

Aims to utilise lightweight, open standards and Web technologies



Data as a service

The central/master hub provides:

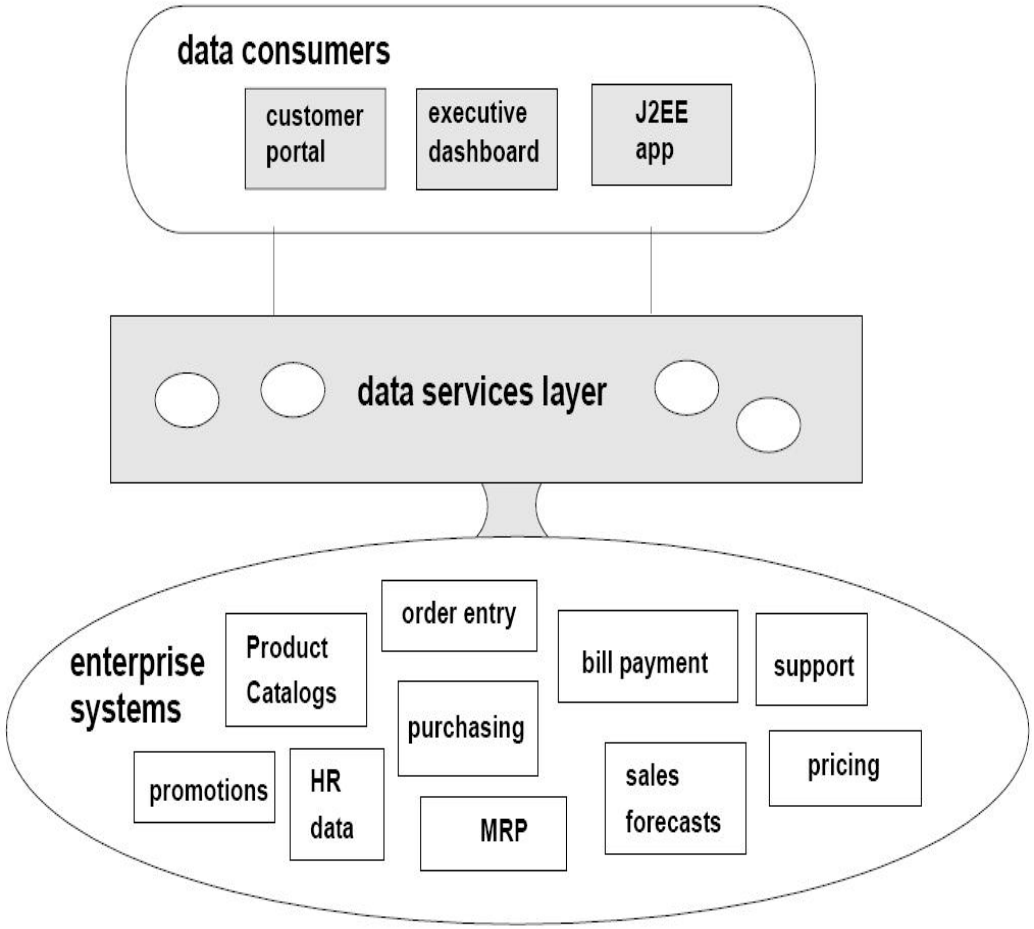
Data model.

Read functions: Returning different representations of the data model

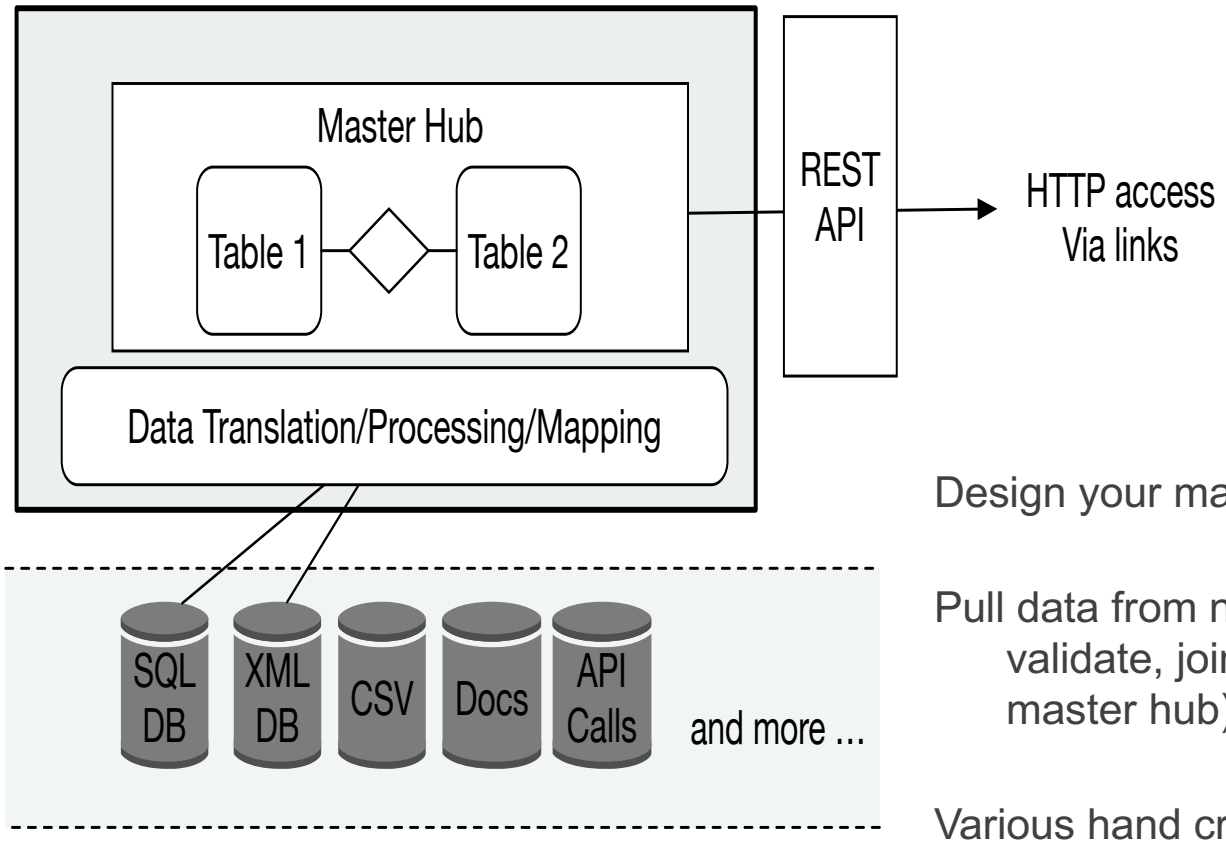
Navigate functions: Returning data from its related data (relationships)

Procedures: Executing side effecting functions in the enterprise.

A submit function: Allowing authorized users to insert, delete, and change back-end data.



Data aggregation/integration



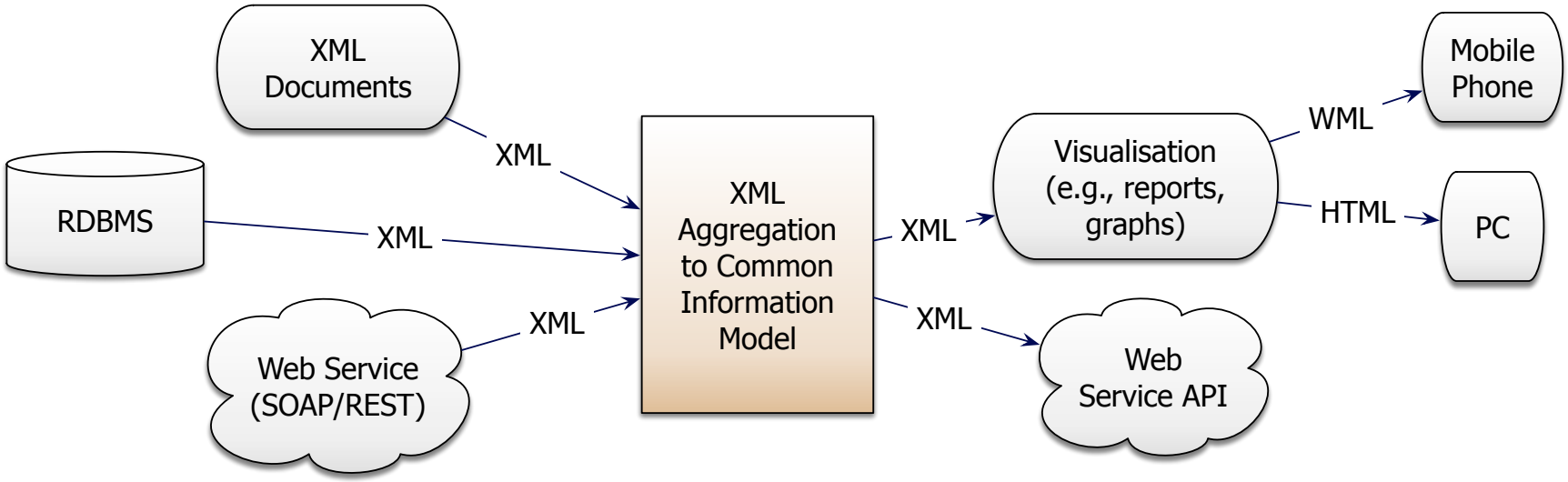
Design your master data hub:

Pull data from necessary sources (clean, validate, join/merge → mapping to your master hub)

Various hand crafted data access skills still necessary (SQL, XML and its family, Web services, JSON, document parsing, etc.)

E.g., a simple data service pattern

Aggregate -> Common Data Model (Storage) -> Publish (Multiple Formats)



Heavy weight support from many vendors

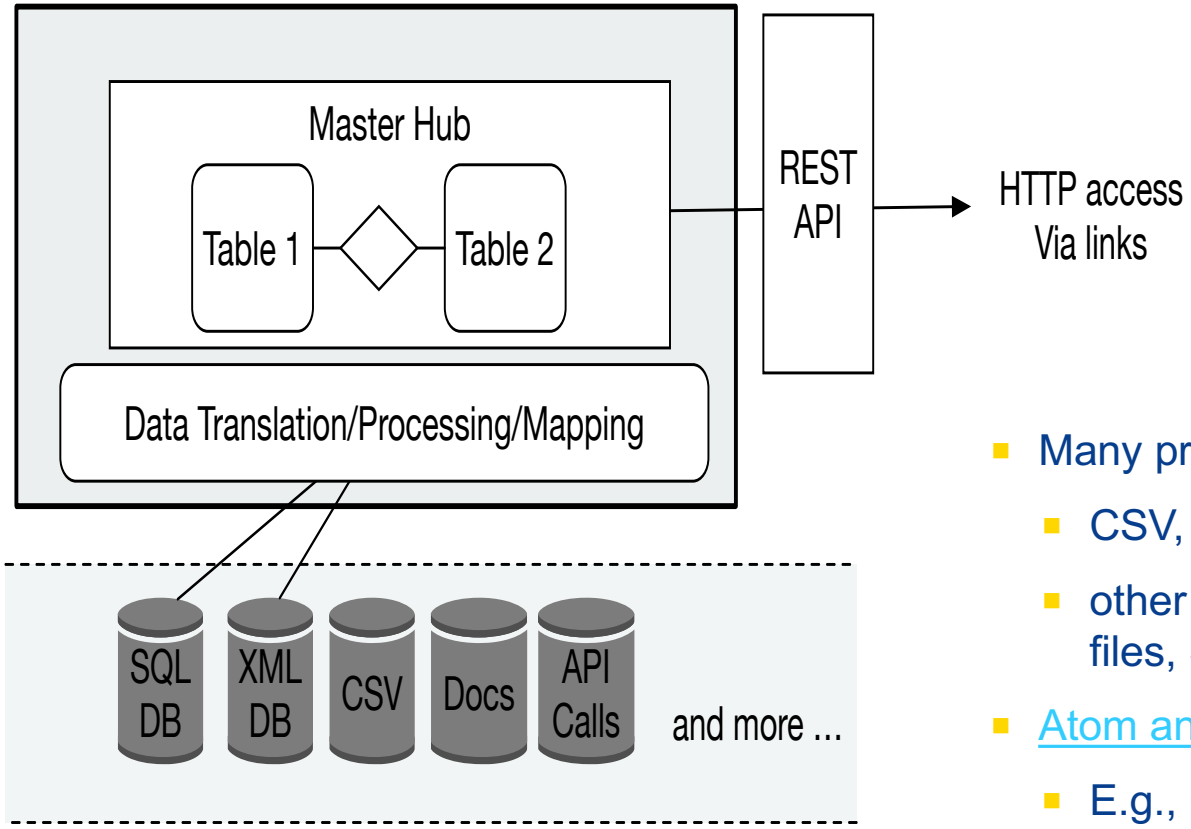
E.g., tool kits

- https://docs.oracle.com/cd/E13167_01/aldsp/docs25/concepts/index.html
- https://docs.microsoft.com/en-us/sql/master-data-services/master-data-services-overview-mds

E.g., big data analytics supporting architecture

- https://www.oracle.com/applications/customer-experience/data-cloud/solutions/data-as-a-service/index.html
- https://www.ibm.com/analytics/us/en/technology/cloud-data-services/#cds-solutions

Data Publication/access interface



- Many presentation formats:
  - CSV, XML, JSON, …
  - other standards (such as shape files, SVG)
- Atom and AtomPub protocol
  - E.g., OData (both for XML, JSON)
- Access design => Navigation and Query
  - REST architecture and Hyperlinks

ATOM and ATOM Publishing Protocol

Atom represents data as lists, called *feeds*. Feeds are made up of one or more timestamped *entries*, which associate document metadata with web content.

Atom Publishing Protocol (AtomPub), a protocol that is built on top of Atom, and which is used for publishing and editing web resources -> Using HTTP actions

The ATOM format (XML)

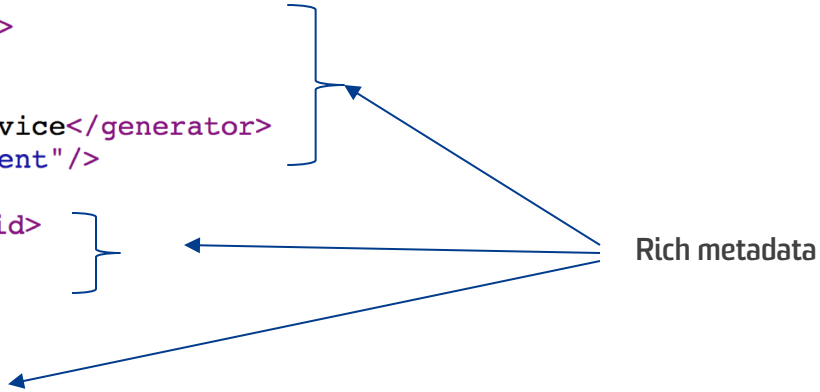
– Atom specification (RFC 4287)

```
▼<feed xmlns="http://www.w3.org/2005/Atom">
   <id>urn:uuid:d0b4f914-30e9-418c-8628-7d9b7815060f</id>
   <title type="text">Recent Orders</title>
   <updated>2009-07-01T12:05:00Z</updated>
   <generator uri="http://restbucks.com/order">Order Service</generator>
   <link rel="self" href="http://restbucks.com/order/recent"/>
  ▼<entry>
     <id>urn:uuid:aa990d44-fce0-4823-a971-d23facc8d7c6</id>
     <title type="text">order</title>
     <updated>2009-07-01T11:58:00Z</updated>
    ▼<author>
       <name>Jenny</name>
     </author>
     <link rel="self" href="http://restbucks.com/order/1"/>
    ▼<content type="application/vnd.restbucks+xml">
      ▼<order xmlns="http://schemas.restbucks.com/order">
        ▼<item>
           <milk>whole</milk>
           <size>small</size>
           <drink>latte</drink>
         </item>
        ▼<item>
           <milk>whole</milk>
           <size>small</size>
           <drink>cappuccino</drink>
         </item>
```

ATOM

The Atom metadata capturing useful business information, such as who took the order and when. The metadata provides friendly descriptions of content, links to other services or resources, and, a means of navigating to other feeds—all in a standard manner.

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>urn:uuid:d0b4f914-30e9-418c-8628-7d9b7815060f</id>
  <title type="text">Recent Orders</title>
  <updated>2009-07-01T12:05:00Z</updated>
  <generator uri="http://restbucks.com/order">Order Service</generator>
  <link rel="self" href="http://restbucks.com/order/recent"/>
  <entry>
    <id>urn:uuid:aa990d44-fce0-4823-a971-d23facc8d7c6</id>
    <title type="text">order</title>
    <updated>2009-07-01T11:58:00Z</updated>
    <author>
      <name>Jenny</name>
    </author>
    <link rel="self" href="http://restbucks.com/order/1"/>
    <content type="application/vnd.restbucks+xml">
      <order xmlns="http://schemas.restbucks.com/order">
        <item>
          <milk>whole</milk>
          <size>small</size>
          <drink>latte</drink>
        </item>
        <item>
          <milk>whole</milk>
          <size>small</size>
          <drink>cappuccino</drink>
        </item>
```



Common usage of ATOM

Syndicating content: an ideal representation format for a publisher distributing content to many consumers.

Representing documents and document-like structures: an ideal choice if your resources are structured like documents – just map the resource's attributes to Atom's metadata elements.

Creating metadata-rich lists of resources: if need to represent ordered lists, such as search results or events. Atom metadata can represent event metadata, thereby establishing an event-oriented processing context for each Atom entry's payload.

Adding metadata to existing resource representations:

- Atom metadata elements can be added to surface information related to a resource (e.g., its author, date created). Attach hypermedia links to existing resource representations by embedding the representation inside an Atom entry and adding one or more <atom:link> elements to the entry.
- The same applies to creating representations for non hypermedia format (such as binary objects). Use the <atom:content> element's src attribute to link to the resource, and specify a media type using the element's type attribute.

ATOM Publishing Protocol

As a publishing protocol, AtomPub provides a standard mechanism for creating and editing resources, and resolving any arising conflicts. It specifies the HTTP idioms that can be used to manipulate published content → Fits with REST API

Terminology: Entry, Feed/Collection, Category Document, Service Document

Basics of AtomPub

- Each resource has a unique identifier
- The resources are well-connected
- Resources share the same interface
- There is no state; requests are atomic
- Follows a resource-oriented architecture

https://www.slideshare.net/benramsey/introduction-to-atompub-web-services

ATOM Publishing Protocol

Feed/Collection (The main resources)

```
▼<feed xmlns="http://www.w3.org/2005/Atom" xmlns:app="http://www.w3.org/2007/app">
  <title>Product Catalog</title>
  <link rel="self" href="http://restbucks.com/product-catalog"/>
  <updated>2010-02-01T13:04:30Z</updated>
  <generator uri="http://restbucks.com/product-catalog">Product Catalog Service</generator>
  <id>urn:uuid:1d0f1a52-31d7-11df-b8ee-f47856d89593</id>
 ▼<entry>
    <title>Fairtrade Roma Coffee Beans</title>
    <id>urn:uuid:7b512808-31d7-11df-aede-127c56d89593</id>
    <updated>2010-01-29T08:22:00Z</updated>
    <app:edited>2010-02-01T13:04:30Z</app:edited>
   ▼<author>
      <name>Product Manager A</name>
    </author>
   ▼<content type="application/vnd.restbucks+xml">
    ▼<product xmlns="http://schemas.restbucks.com/product">
       <name>Fairtrade Roma Coffee Beans</name>
       <size>1kg</size>
       <price>10</price>
     </product>
    </content>
    <link rel="edit" href="http://restbucks.com/product-catalog/1234"/>
  </entry>
 ▼<entry>
    <title>Fairtrade Roma Coffee Beans</title>
    <id>urn:uuid:08acdbfe-31db-11df-9fa6-839856d89593</id>
    <updated>2010-01-29T08:22:00Z</updated>
    <app:edited>2010-01-29T08:22:00Z</app:edited>
   ▼<author>
      <name>Product Manager A</name>
    </author>
    <summary type="text">Fairtrade Roma Coffee Beans image</summary>
    <content type="image/png" src="http://restbucks.com/product-catalog/fairtrade_roma.png"/>
    <link rel="edit-media" href="http://restbucks.com/product-catalog/fairtrade_roma.png"/>
    <link rel="edit" href="http://restbucks.com/product-catalog/5555"/>
  </entry>
```

ATOM Publishing Protocol

Category Documents. Category documents contain lists of categories for categorizing collections and members. A category list can be fixed, meaning it's a closed set, or left open, allowing for subsequent extension.

Category documents have its own media type of application/atomcat+xml.

```
<categories xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom"
 scheme="http://restbucks.com/product-catalog/categories/status" fixed="yes">
   <atom:category term="new"/>
   <atom:category term="updated"/>
   <atom:category term="deleted"/>
 </categories>
```

ATOM Publishing Protocol

Service Documents. A service document acts as a well-known entry point into the collections hosted by a service. From a service document, a client can navigate to the collections provided by the service

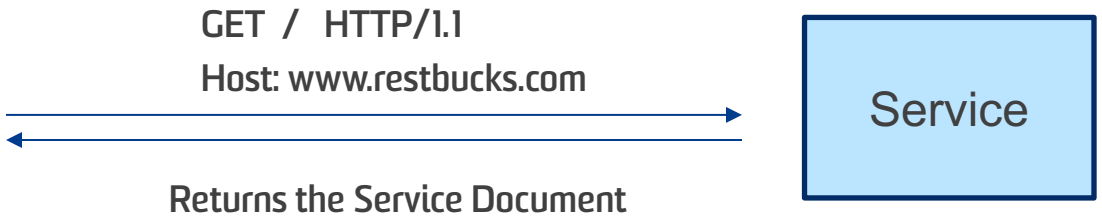
Service documents have its own media type of application/atomsvc+xml.

```
▼<service xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom">
 ▼<workspace>
    <atom:title>Product Catalog</atom:title>
   ▼<collection href="http://restbucks.com/product-catalog/products">
      <atom:title>Products</atom:title>
      <accept>application/atom+xml;type=entry</accept>
      <categories href="http://restbucks.com/product-catalog/categories/status"/>
    </collection>
   ▼<collection href="http://restbucks.com/product-catalog/promotions">
      <atom:title>Promotions</atom:title>
      <accept>application/atom+xml;type=entry</accept>
      <accept>image/png</accept>
      <accept>image/gif</accept>
      <categories href="http://restbucks.com/product-catalog/categories/status"/>
      <categories href="http://restbucks.com/product-catalog/categories/scope"/>
     ▼<categories scheme="http://restbucks.com/product-catalog/categories/origin" fixed="yes">
        <atom:category term="in-house"/>
        <atom:category term="partner"/>
      </categories>
    </collection>
  </workspace>
</service>
```

AtomPub in Action

If you were to implement the Assignment 1 with AtomPub as main interaction protocol …

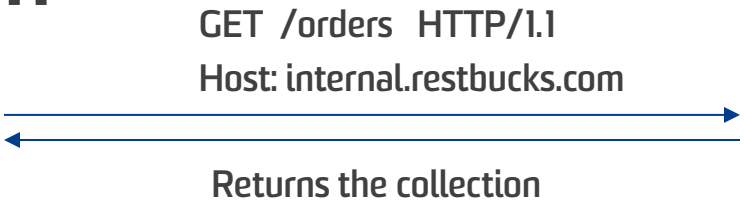
1. Service Discovery




```
▼<service xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom">
  ▶<workspace>...</workspace>
  ▼<workspace>
     <atom:title>Product Sales</atom:title>
    ▼<collection href="http://internal.restbucks.com/orders/">
       <atom:title>Orders</atom:title>
       <accept>application/atom+xml;type=entry</accept>
      ▼<categories scheme="http://internal.restbucks.com/orders/status" fixed="yes">
         <atom:category term="open"/>
         <atom:category term="cancelled"/>
       </categories>
     </collection>
   </workspace>
 </service>
```

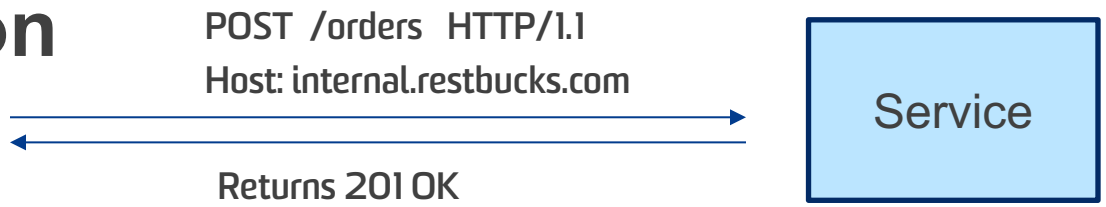
AtomPub in Action

2. Get a collection of the data items




```
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:app="http://www.w3.org/2007/app">
  <title>Order Fulfillment</title>
  <link rel="self" href="http://internal.restbucks.com/orders"/>
  <updated>2017-03-29T13:00:30Z</updated>
  <generator uri="http://internal.restbucks.com/orders">Order Fulfillment Service</generator>
  <id>urn:uuid:6d2992ae-ec8a-4dac-91b3-d452186ea409</id>
▼<app:collection href="http://internal.restbucks.com/orders">
    <title>Order Fulfillment Service</title>
    <app:accept>application/atom+xml;type=entry</app:accept>
  </app:collection>
▼<entry>
    <title>order</title>
    <id>urn:uuid:fc2d3d42-7198-4c59-a936-b9b870ef8469</id>
    <updated>2017-03-29T13:00:30Z</updated>
    <app:edited>2017-03-29T13:00:30Z</app:edited>
  ▼<author>
      <name>Cashier</name>
    </author>
    <link rel="self" href="http://internal.restbucks.com/orders/1234"/>
    <link rel="edit" href="http://internal.restbucks.com/orders/1234"/>
    <content type="application/atom+xml">...</content>
  </entry>
  <entry>...</entry>
</feed>
```

AtomPub in Action



3. Creating an order

```
POST /orders HTTP/1.1
Host: internal.restbucks.com
Content-Type: application/atom+xml
Content-Length: ...

<entry xmlns="http://www.w3.org/2005/Atom">
  <title>order</title>
  <author>
    <name>Cashier</name>
  </author>
  <content type="application/atom+xml;entry">
    <order xmlns="http://schemas.restbucks.com/order">
      <consume-at>takeAway</consume-at>
      <items>
        <item>
          <name>latte</name>
          <quantity>1</quantity>
          <size>small</size>
        </item>
      </items>
    </order>
  </content>
</entry>
```

```
HTTP/1.1 201 Created
Date: ...
Content-Length: ...
Content-Type: application/atom+xml;type=entry;charset="utf-8"
Location: http://internal.restbucks.com/orders/9876
Content-Location: http://internal.restbucks.com/orders/9876
ETag: "3f2b06f7"

<entry xmlns="http://www.w3.org/2005/Atom"
       xmlns:app="http://www.w3.org/2007/app">
  <title>order</title>
  <id>urn:uuid:e557e51b-c994-44ef-b06d-5331246cccbe</id>
  <author>
    <name>Cashier</name>
  </author>
  <link rel="self" href="http://internal.restbucks.com/fulfillment/9876"/>
  <link rel="edit" href="http://internal.restbucks.com/fulfillment/9876"/>
  <content type="application/atom+xml;type=entry;">
    ...
  </content>
</entry>
```

AtomPub in Action

PUT updates an <Entry>

DELETE removes an <Entry>

So … basically all REST-based interface and its principle applies here

(plus some protocol specific constraints)

Obviously … some of the CHANGE operations must be accompanied by authentication and authorisation.

ATOM feeds: ATOM metadata + your data model (inside <content>)

OData – ATOM-based data publication format

OData (a short name for Open Data protocol)

- A standard API for data access (managed by the OASIS group)
- Based on Web data feed standard ATOM and AtomPub
- One can say … it is an ATOM-based implementation of Data Service API
- Also, one can say … OData is:
  - JDBC/ODBC on the Web
  - SQL query over the Web using URL

OData - *A Standard API for Data Access, by* Pat Patterson (Salesforce.com) on slideshare.net
OData and impact on API design by Apigee Webinar – on slideshare.net

What problem does it solve?

Restful APIs are great, but …

REST is a *style*, not a *standard*

- RESTful, RESTlike, RESTish

URL parameters?

- e.g. retrieve only a subset of properties
- e.g., Retrieve a set of records via a query?

No easy or standardized ways to access the metadata (e.g., table catalogs)

www.odata.org

OData is a standardized protocol for creating and consuming data APIs. OData builds on core protocols like HTTP and commonly accepted methodologies like REST. The result is a uniform way to expose full-featured data APIs.

Proposed by Microsoft in 2009

Standardized by OASIS in 2014

OData supports XML (Atom) and JSON

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom" xml
/ado/2007/08/dataservices" xmlns:m="http://sch
aservices/metadata" xmlns:georss="http://www.g
//www.opengis.net/gml" xml:base="http://servic
  <id>http://services.odata.org/V3/OData/OData
  <category term="ODataDemo.Product" scheme="h
07/08/dataservices/scheme"/>
  <link rel="edit" title="Product" href="Produ
  <link rel="http://schemas.microsoft.com/ado/
ories" type="application/atom+xml;type=feed" t
)/Categories"/>
  <link rel="http://schemas.microsoft.com/ado/
ier" type="application/atom+xml;type=entry" ti
upplier"/>
  <link rel="http://schemas.microsoft.com/ado/
ctDetail" type="application/atom+xml;type=entr
ducts(1)/ProductDetail"/>
  <title type="text">Milk</title>
  <summary type="text">Low fat milk</summary>
  <updated>2014-09-15T21:02:15Z</updated>
```

```
{
    "value": [
        {
            "Description": "Whole grain bread",
            "DiscontinuedDate": null,
            "ID": 0,
            "Name": "Bread",
            "Price": 2.5,
            "Rating": 4,
            "ReleaseDate": "1992-01-01T00:00:00Z"
        },
        {
            "Description": "Low fat milk",
            "DiscontinuedDate": null,
            "ID": 1,
            "Name": "Milk",
            "Price": 3.5,
            "Rating": 3,
            "ReleaseDate": "1995-10-01T00:00:00Z"
        },
```

OData Requests and Responses

REST Interface (HTTP access links)

- GET, POST, PUT/PATCH, DELETE

- Supports all types of metadata prescribed by ATOM and ATOMPUB

- Service Document (JSON/XML): http://services.odata.org/V4/OData/OData.svc/
- Metadata (XML only): http://services.odata.org/V4/OData/OData.svc/$metadata
- Let‘s pick one entity and explore …
- http://services.odata.org/V4/OData/OData.svc/Products
- http://services.odata.org/V4/OData/OData.svc/Products?$filter=Rating+eq+3&$select=Rating,+Name

OData – SQL over URL

SQL Query	OData Request
select * from products where id = 1	/Products(1)
select * from products where name = ‘Milk’	/Products?$filter=name eq ‘Milk’
select name from products	/Products?$select=name
select * from products order by name	/Products?$orderby=name
select * from products offset 10 limit 10	/Products?$top=10&$skip=10
select * from prices r, products p where r.id = p.id (* sort of)	/Products(1)?$expand=Prices

OData – Industry support …

OData:

- Standardizes data-centric web services
- Exposes Data *and* Metadata
- JSON or XML (Atom/AtomPub) representation over HTTP

Wide industry support

OData servers:

Microsoft

- Microsoft SQL Server
- Windows Azure Active Directory

SAP NetWeaver

Salesforce Platform Connect

IBM WebSphere

RedHat JBoss

- OData libraries (www.odata.org/libraries)
  - Java, .Net
  - JavaScript, Node.js
  - Objective-C
  - Python, Ruby, PHP
  - C++