2. DATA MODELS AND QUERY LANGUAGES

1. Relational Model Versus Document Model.
   * The Birth of NoSQL
   * The Object-Relational Mismatch.
   * Many-to-One and Many-to-Many Relationships.
   * Are Document Databases Repeating History?
   * Relational Versus Document Databases Today.

2. Query Languages for Data
   * Declarative Queries on the Web.
   * MapReduce Querying.

3. Graph-Like Data Models.
   * Property Graphs.
   * The Cypher Query Language.
   * Graph Queries in SQL.
   * Triple-Stores and SPARQL.
   * The Foundation: Datalog.

* 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?

① Application developer: model real-world in terms of objects & data structures, and APIs that manipulate those data structures.
② Store those data structures: general purpose data model, such as JSON & XML documents, tables in a relational DB, & a graph model.
③ DB engineers: represent JSON/XML/relational/graph data in terms of bytes in memory, on disk, & on a network. The representation may allow the data to be queried, searched, manipulated, and processed.
④ Hardware engineers: represent bytes in terms of currents, magnetic fields etc.

Scanned with CamScanner

General-Purpose data models :-

for storage and querying:

Relational model, Document model, Graph-based data models.

Query languages and Use Cases :-

1) RELATIONAL MODEL VERSUS DOCUMENT MODEL

Relational Model – proposed by Edgar Codd in 1970
↳ data is organized into :
* relations (called tables in SQL)
↳ Each relation is an unordered collection of tuples (rows in SQL).

mid 1980s: (popular) RDMSs and SQL to store & query data with structure.
1960s & 1970s: "business data processing" on mainframe computers.
Today: "transaction processing" –
- entering sales or banking transactions.
- airline reservations.
- stock-keeping in warehouses.

"batch processing"
- customer invoicing
- payroll
- reporting.

"The goal of the relational model was to hide that implementation detail (internal representation of data in DB) behind a cleaner interface."

Scanned with CamScanner

1970s and early 1980s network model and hierarchical model were main alternatives, but relational model dominated.
late 1980s and early 1990s Object DBs
early 2000s XML DBs.

* THE BIRTH OF NoSQL :-

2010s: NoSQL.

"Not Only SQL"

Driving forces to adopt NoSQL:

* Greater scalability than RDB's, including very large datasets & very high write throughput.
* Preference for free and open source software over commercial DB products.
* Specialized query operations that are not well supported by relational model.
* Restrictiveness of relational schemas and a desire for a more dynamic and expressive data model.

RDBs will continue to be used alongside Non RBs - polyglot persistence.

* THE OBJECT - RELATIONAL MISMATCH :-

Object oriented programming language ⚡ Relational tables.

↳ translation layer is required due to impedance mismatch.

ORM frameworks like Active Record & Hibernate reduce the boilerplate code required for this translation layer.

Scanned with CamScanner

Resume in Relational schema:-

users table:-

user_id	first_name	last_name	summary	region_id
251	Bill	Gates	Co-chair...	us:91

industry_id	photo_id
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...
457	251	Co-founder, Chairman.	Microsoft.

education table:-

id	user_id	school_name	start	end
807	251	Harvard Uni.	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

Resume:-

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Cochair of ...

Chairman, Microsoft.....

Experience

Cochair - .... - 2000 - Present

Co-founder - .... - 1975 - Present

Education

Scanned with CamScanner

Harvard University
1973 - 1975
Lakeside School, Seattle

Contact Info
Blog: thegatenotes.com.
Twitter: @BillGates.

-fig., Representing a LinkedIn profile using a relational schema.

* One-to-many relationship: user to ~~job~~ positions, education, contact represented in various ways:

① 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.

② later versions of SQL
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. Supported by Oracle, IBM DB2, MS SQL Server, Postgre SQL

JSON datatype supported by IBM DB2, MySQL, Postgre SQL.

③ encode jobs/positions, education, contact into JSON & XML document.
store it in a text column in DB and let the application interpret its structure and content. In this setup, you typically cannot use the DB to query for values inside that encoded column.

"JSON representation can be quite appropriate."
"Document-oriented DBs like MongoDB, RethinkDB, CouchDB, Espresso support this data model."

Scanned with CamScanner

Representing a LinkedIn profile as a JSON document:

```
{
    "user_id":      251,
    "first_name":   "Bill",
    ....
    ....
    "region_id":    "us:91",
    ....
    ....
    "positions": [
        {"job_title": "Co-chair", "organization": "Bill & ..."},
        {"job_title": ....... }
    ],
    "education": [
        {"school_name": "Harvard University", "start": 1973, "end": 1975},
        { ..... }
    ],
    "contact_info": {
        "blog":    "http://thegatsnotes.com",
        "twitter": "http://twitter.com/BillGates"
    }
}
```

physically close, minimal data movement

JSON representation has better "locality" than multi-table schema.

One query sufficient in JSON representation
whereas in relational, multiple queries & complex JOINs required.

These one-to-many relationships imply a tree structure in the data, and the JSON representation makes the tree structure explicit.

Scanned with CamScanner

M T W T F S S
Page No.:
Date:
YOUVA

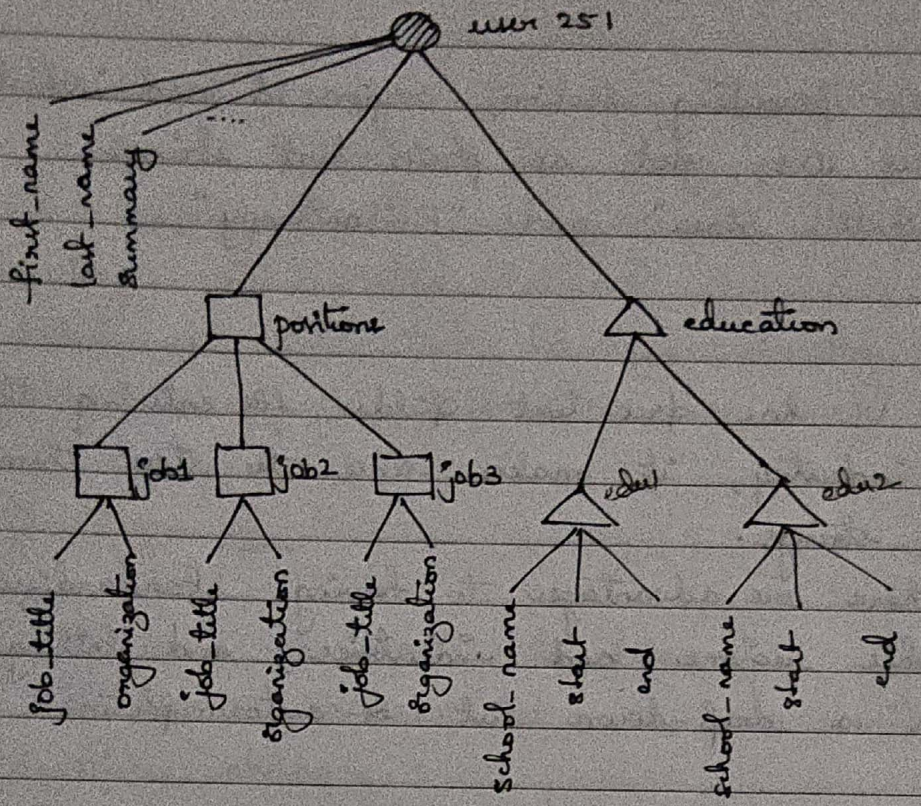


Fig., One-to-many relationships forming a tree structure

✳ Many-to-One and Many-to-Many Relationships

Scanned with CamScanner

* MANY-TO-ONE AND MANY-TO-MANY RELATIONSHIPS

In the preceding section, region_id and industry_id are given as IDs, not as plain-text strings "Greater Seattle Area" and "Philanthropy".

Reason:

If the UI has free-text fields for entering the region and the industry, it makes sense to store them as plain-text strings.

But there are advantages to having standardized lists of geographic regions and industries, and letting users choose from a drop-down list & autocompleter.

* 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.

Scanned with CamScanner

M T W T F S S
Page No.
Date
YOUVA

ID & a text string to be stored?

When you use an ID, the information that is meaningful to humans is stored in only one place, and everything that refers to it uses an ID (which only has meaning within the database).

When you store text directly, you are duplicating the human-meaningful information in every record that uses it.

Advantages of ID :-

ID has no meaning to humans.

So ID can remain the same, even if the information it identifies changes.

Anything that is meaningful to humans may need to change sometime in the future — and if that information is duplicated, all the redundant copies need to be updated. That incurs write overloads, and risks inconsistencies (where some copies of the information are updated but others aren't). Removing such duplication is the key idea behind normalization in databases.

If you're duplicating values that could be stored in just one place, the schema is not normalized — rule of thumb.

Many-to-one relationships don't fit nicely into the document model :- many people live in 1 particular region.
many people work in 1 particular industry.

At a point of time, joins aren't supported in MongoDB and only supported in predeclared views in CouchDB but not supported in RethinkDB.

Scanned with CamScanner

In case of databases that does not support joins, the work of making the join is shifted from the database to the application.

Even if the initial version of an application fits well in a join-free document model, data has a tendency of becoming more interconnected as features are added to applications.

Changes to resume example :-

1) Organizations and schools as entities :- entities have reference (like) (fig 1)
2) Recommendations - one user can write recommendation to another user.

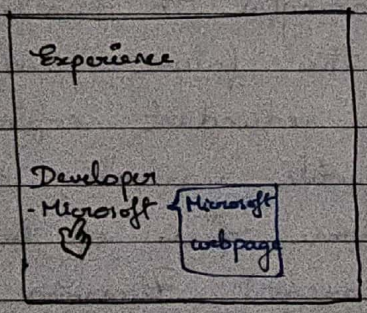


fig 1, LinkedIn Resume.

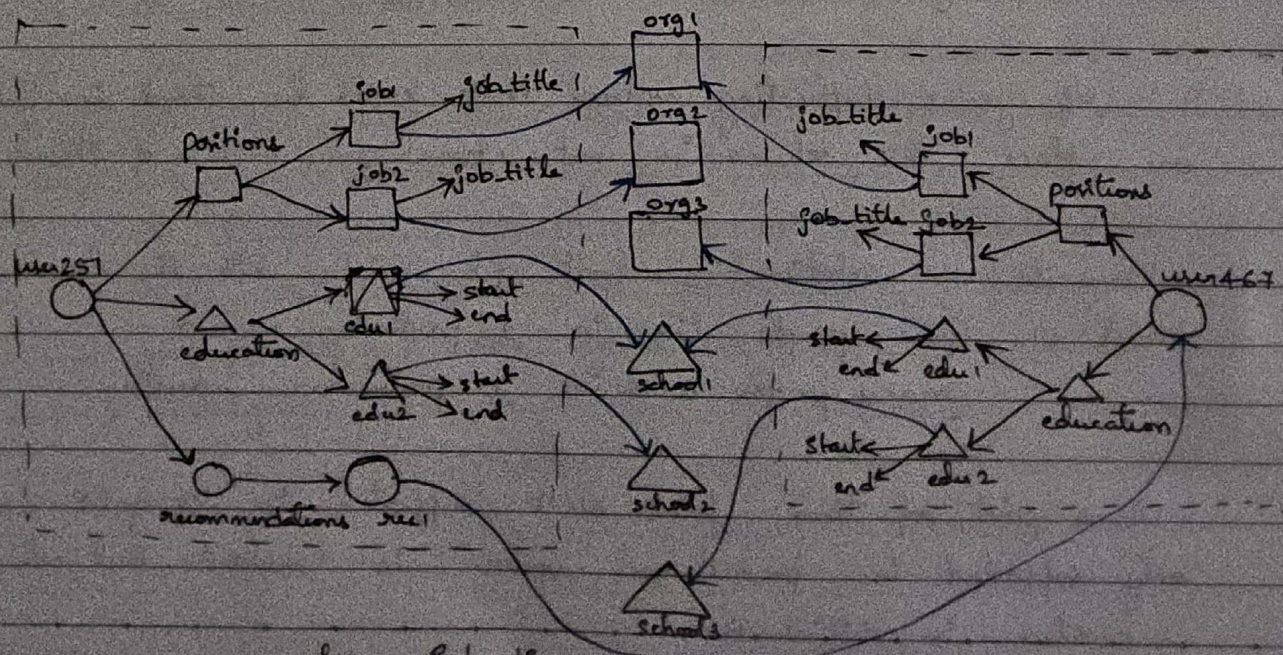


fig 2, Extending resumes with many-to-many relationships

Scanned with CamScanner

* ARE DOCUMENT DATABASES REPEATING HISTORY?

many-to-many relationships and joins are routinely used in relational databases.

Earliest computerized database systems :-

* IBM's Information Management System (IMS)
* IMS used a data model called the hierarchical model which has some remarkable similarities to the JSON model used by document databases.
* Proposed solutions to solve the limitations of the hierarchical model:
  - relational model
  - network model.

The network model :-

* standardized by a committee called CODASYL and also called CODASYL model.
* CODASYL - Conference on Data Systems Languages.
* The CODASYL model was a generalization of the hierarchical model.
* In the tree structure of the hierarchical model, every record has exactly one parent;
* In the network model, a record could have multiple parents.
* access path :- a record can be accessed on a path from root record, along the chains of links.
* This was like navigating around an n-dimensional data space.

Scanned with CamScanner

The relational model :-

* a collection of tuples (rows).
* In a relational database, the query optimizer automatically decides which parts of the query to execute in which order, and which indexes to use. This is similar to "access path" in network model.
* Advantages :-
  i) The above choices of query order and indexes are made by query optimizer automatically, not by the application developer.
  ii) If you want to query data in new ways, you can declare a new index and queries will automatically use whichever indexes are most appropriate. No need to change queries for new index. Thus adding new features is easy.

Comparison to document databases :-

* one-to-many relationships : * document databases using hierarchical model
* many-to-one & many-to-many relationships :
  * relational — foreign key
  * document — document reference

Scanned with CamScanner

* RELATIONAL VERSUS DOCUMENT DATABASES TODAY

* fault-tolerance
* handling of concurrency } later chapters
* differences in data model.

document data model	relational model
- schema flexibility	- better support for joins and many-to-one & many-to-many relationships.
- better performance due to locality.	
- closer to the data structures used by the application.	

1) Which data model leads to simpler application code?
2) Schema flexibility in the document model.
3) Data locality for queries
Convergence of document and relational databases.

1) Which data model leads to simpler application code?

* If data is in document-like structure (i.e., a tree of one-to-many relationships, where tree is loaded at once), then it's probably a good idea to use a document model.
* shredding - splitting a document-like structure into multiple tables - can lead to complicated application code.
* limitations - you can't refer directly to a nested item (much like an access path in the hierarchical model). Unless documents are not too deeply nested, that is not usually a problem.
* For many-to-many relationships, it's possible to reduce the need for joins by denormalizing, but then the application

Scanned with CamScanner

code needs to do additional work by making multiple requests to the database, but that also moves complexity into the application and is usually slower than a join performed by specialized code inside the database.

* Which data model depends on kind of relationships that exist between data items. For highly interconnected data, the document model is awkward, the relational model is acceptable, and graph models are the most natural.

2) Schema flexibility in the document model

Document databases are called schemaless, but that's misleading, as the code that reads the data usually assumes some kind of structure - i.e, there is an implicit schema, but it is not enforced by the database. - (schema-on-read) - the structure of the data is implicit and only interpreted when the data is read. In contrast, (schema-on-write) - the traditional approach of relational databases, where the schema is explicit and the database ensures all data written data conforms to it.

* Schema-on-read : dynamic (runtime) type checking
* Schema-on-write : static (compile-time) type checking w.r.t programming languages.

Scanned with CamScanner

Difference between schema-on-read and schema-on-write

when application wants to change the format of its data.

Eg:- currently user name is in a single field, and instead want to store first name and last name separately.

* In document database - you would just start writing new documents with the new fields and you would have application code that handles the case when old documents are read.

```
Ex:- 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];
     }
```

* In a "statically typed" database schema, - you would perform a migration along the lines of:

```
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
```

Schema changes are fast in few databases and slow especially on large tables.

Schema-on-read approach is advantageous if the items in the collection don't all have the same structure for some reason (i.e, the data is heterogeneous) because

* There are many different types of objects, and it is not practical to put each type of object in its own table.
* The structure of the data is determined by external systems over which you have no control and which may change at any time.

But in cases where all records are expected to have the same structure, schemas are a useful mechanism.

3) Data locality for queries

* A document is usually stored as a single continuous string, encoded as JSON, XML, or a binary variant thereof (such as MongoDB's BSON).
* If your application often needs to access the entire document (for example, to render it on a web page), there is a performance advantage to this storage locality.
* If data is split across multiple tables, multiple index lookups are required to retrieve it all, which may require more disk seeks and take more time.

* The locality advantage only applies if you need large parts of the document at the same time.
* The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents.
* On updates to a document, the entire document usually needs to be rewritten - only modifications that don't change the encoded size of a document can easily be performed in place.

Scanned with CamScanner

* For these reasons, it is generally recommended that you keep documents fairly small and avoid writes that increase the size of a document.
* These performance limitations significantly reduce the set of situations in which document databases are useful.

* Idea of grouping related data together for locality is not limited to the document model.

Eg:-

Google's Spanner database offers locality properties in a relational data model, by allowing the schema to declare that a table's rows should be interleaved (nested) within a parent table.

Oracle allows locality by using a feature called multi-table index cluster tables.

The column-family concept in a Bigtable data model (used in Cassandra and HBase) has a similar purpose of managing locality.

4) Convergence of document and relational databases :-

* Relational DBMS's supported XML (other than MySQL)
* abilities to make local modifications to XML documents and the ability to index and query inside XML documents
* this allows applications to use data models very similar to what they would do when using a document database.

On the document database side, RethinkDB supports relational-like joins in its query language.

Some MongoDB drivers automatically resolve database references (likely to be slower than join performed in the DB since it requires additional network round-trips)

*Postgres, MySQL, IBM DB2 have similar support for JSON docs

Scanned with CamScanner

* QUERY LANGUAGES FOR DATA

* Declarative and Imperative Query language :-

SQL - declarative.

IMS & CODASYL - imperative.

Example of imperative :-

If you have a list of animal species, you want to return only sharks : (in programming language)

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

Example of declarative :-

(in relational algebra)

$$sharks = \sigma_{family = \text{"Sharks"}}(animals)$$

SQL followed the structure of relational algebra closely.

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

Scanned with CamScanner

* In imperative language tells the computer to perform certain operations in a certain order.

* In declarative query language, SQL & relational algebra, you just specify the pattern of the data you want - what conditions the result must meet and how you want the data to be transformed - but not how to achieve the goal. It is upto query optimizer to decide which indexes & queries & order of execution of various parts of the query.

Advantage of declarative over imperative :-

* Declarative query language hides implementation details of the database engine, which makes it possible for the database system to introduce performance improvements without requiring any changes to queries.
* In the imperative code shown, the list of animals appears in a particular order. If the records are moved around, then the order will be changed
* In the SQL code, there is no guarantee of any particular ordering and so it doesn't mind if the order changes.
* Declarative languages are advantageous for parallel execution, because they specify only the pattern of the results, not the algorithm that is used to determine the results.

Scanned with CamScanner

1. Declarative Queries on the Web :-

Declarative and Imperative approaches on a web browser:-

Example: Website of animals in the ocean.
The user is currently viewing the page on sharks, so you mark the navigation item "Sharks" as currently selected, like this :

CSS:-
```
<ul>
  <li class = "selected">
    <p> Sharks </p>
    <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>
```

Now, you want the title of the currently selected page to have a blue background.

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

Scanned with CamScanner

XSL :-

```
<xsl:template match = "li[@class = 'selected']/p" >
    <fo:block background-color = "blue" >
        <xsl:apply-templates />
    </fo:block>
</xsl:template>
```

CSS and XSL are both declarative languages for specifying the styling of a document.

Imperative approach, in JavaScript, using the Document Object Model (DOM) API :

```
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");
            }
        }
    }
}
```

Scanned with CamScanner

* Drawbacks :-

① If 'selected' class is removed, the blue color won't be removed, even if the code is re-run.
② much longer and harder to understand than the CSS and XSL equivalents.
③ If you want to take advantage of a new API, such as document.getElementsByClassName("selected") or even document.evaluate() - which may improve performance of CSS and XPath without breaking compatibility.

So, declarative CSS styling is much better than manipulating styles imperatively in JavaScript, in web browser.

Similarly, in databases, declarative query languages like SQL turned out to be much better than imperative query APIs.

Scanned with CamScanner

MapReduce Querying :-

MapReduce is a programming model for processing large amounts of data in bulk across many machines, popularized by Google.

A limited form of MapReduce is supported by some NoSQL datastores, including MongoDB and CouchDB, as a mechanism for performing read-only queries across many documents.

MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between: the logic of the query is expressed with snippets of code, which are called repeatedly by the processing framework. It is based on the map (also known as collect) and reduce (also known as fold or inject) functions that exist in many functional programming languages.

Example :-

Marine biologist :- Adds an observation record to your database every time you see animals in the ocean. Now wants to generate a report saying how many sharks are sighted per month.

Query in PostgreSQL looks like :-

Scanned with CamScanner

```
SELECT date_trunc('month', observation_timestamp)
AS observation_month①, sum(num_animals) AS total_animals
FROM observations
WHERE family = 'sharks'
GROUP BY observation_month;
```

① The date_trunc('month', timestamp) function determines the calendar month containing timestamp, and returns another timestamp representing the beginning of that month. In other words, it rounds a timestamp down to the nearest month.

This query first filters the observations to only show species in the 'Sharks' family, then groups the observations by the calendar month in which they occurred, and finally adds up the number of animals seen in all observations in that month.

Query in MongoDB with MapReduce feature :-

```
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
    },
```

Scanned with CamScanner

```
{
    query: { family: "sharks" },  ①
    out: "monthlySharkReport"  ⑥
}
);
```

① The filter to consider only shark species can be specified declaratively (this is a MongoDB-specific extension to MapReduce).

② The JavaScript function 'map' is called once for every document that matches 'query', with 'this' set to the document object.

③ The 'map' function emits a key (a string consisting of year and month, such as "2013-12" or "2014-1") and a value (the number of animals in that observation).

④ The key-value pairs emitted by 'map' are grouped by key. For all key-value pairs with the same key (i.e., the same month and year), the 'reduce' function is called once.

⑤ The 'reduce' function adds up the number of animals from all observations in a particular month.

⑥ The final output is written to the collection 'monthlySharkReport'.

Scanned with CamScanner

* Example:-

the 'observations' collection contains these two documents:

```
{
    observationTimestamp: Date.parse("Mon, 25 Dec 1995
                                      12:34:56 GMT"),
    family: "sharks",
    species: "Carcharodon carcharias",
    numAnimals: 3
}
{
    observationTimestamp: Date.parse("Tue, 12 Dec 1995
                                      16:17:18 GMT"),
    family: "sharks",
    species: "Carcharias taurus",
    numAnimals: 4
}
```

* The 'map' function would be called once for each document, resulting in emit("1995-12", 3) and emit("1995-12", 4). Subsequently, the 'reduce' function would be called with reduce("1995-12", [3, 4]), returning 7.

* The 'map' and 'reduce' functions are 'pure' functions, which means they only use the data that is passed to them as input, they cannot perform additional database queries, they must not have any side effects. These restrictions allow the database to run the functions anywhere, in any order, and rerun them on failure. - an advantage of MapReduce

Scanned with CamScanner

In the previous code, we used 2 javascript codes in 'mapReduce'. This is often harder than writing a single query. Moreover, a declarative query language offers more opportunities for a query optimizer to improve the performance of a query.

For these reasons, MongoDB 2.2 added support for a declarative query language called the aggregation pipeline.

```
db.observations.aggregate([
    { $match: { family: "sharks" } },
    { $group: {
        _id: {
            year: { $year: "$observationTimestamp" },
            month: { $month: "$observationTimestamp" }
        },
        totalAnimals: { $sum: "$numAnimals" }
    }
}
]);
```

* GRAPH - LIKE DATA MODELS.

one-to-many → document models.
many-to-many → SQL models.

But as many-to-many relationships become very common and connections become more complex, it's better to model data as a graph.

vertices & nodes & entities.
edges & relationships & arcs.

Examples :-

① Social graphs : Vertices are people,
Edges indicate which people know each other.

② The web graph : Vertices are web pages.
Edges indicate HTML links to other pages.

③ Road & rail networks : Vertices are junctions
Edges represent the roads or railway lines between them.

In these examples, vertices are just homogeneous data. But usage of graphs is just not limited to that.

Example :-

① Facebook : vertices are people, locations, events, checkins, comments;
edges are which people are friends with each other, which checkin happened in which location, who commented on which post etc.

Scanned with CamScanner

Example :- Two people, Lucy from Idaho and Alain from Beaune, France. They are married and living in London.

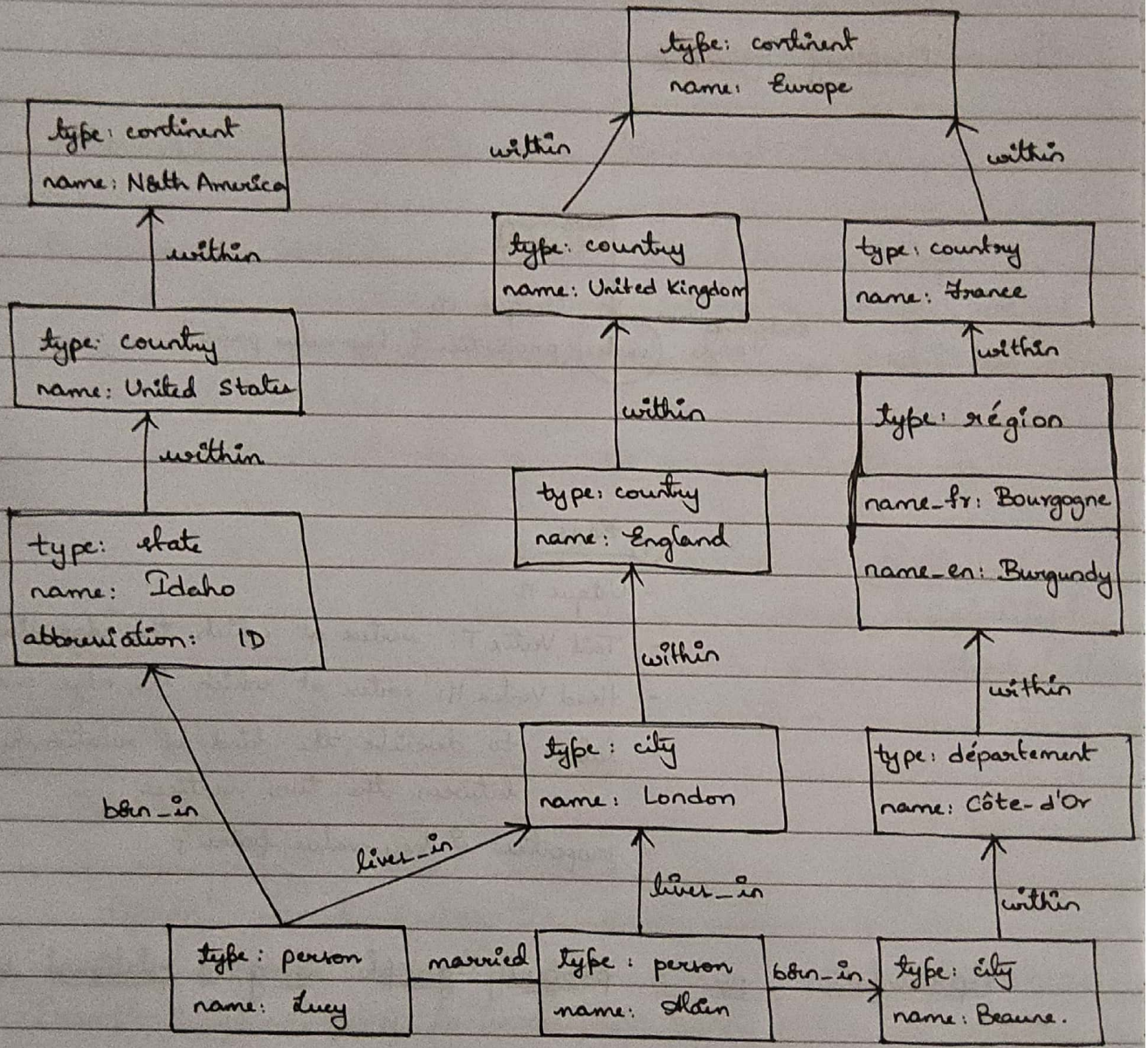


fig., Example of graph-structured data.

* Ways of structuring and querying data in graphs :-

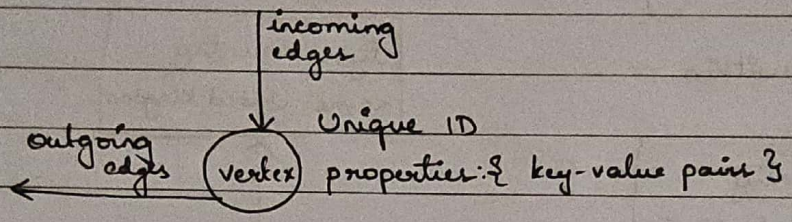
① property graph model (by Neo4j, Titan, Infinite Graph)
② triple-store model (by Datomic, Allegro Graph & others)

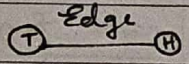
* Declarative query languages for graphs :- Cypher, SPARQL, Datalog.

Scanned with CamScanner

* Imperative graph query languages :- Gremlin
* Graph processing frameworks :- Pregel.

1. Property Graphs :-





- Unique ID
- Tail Vertex T: vertex at which the edge starts.
- Head Vertex H: vertex at which the edge ends.
- label: to describe the kind of relationship between the two vertices.
- properties: { key-value pairs }

Representation of a property graph using a relational schema:-

```
CREATE TABLE vertices (
        vertex_id integer PRIMARY KEY,
        properties json
);
CREATE TABLE edges (
        edge_id integer PRIMARY KEY,
        tail_vertex integer REFERENCES vertices (vertex_id),
        head_vertex integer REFERENCES vertices (vertex_id),
```

Scanned with CamScanner

```
    label text,
    properties jsonb
);
CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

Advantages of graphs over Relational DB :-

* Graphs provide flexibility for data modeling.
* In the previous figure, following things would have been difficult to express in a traditional relational schema :
  - different kinds of regional structures in different countries.
  - country within a country - quirks of history
  - varying granularity of data - Lucy's residence is specified as a city, whereas her birth place in level of state.
* Graphs also are good at extensibility.

The Cypher Query Language :-

'Cypher' is a declarative query language for property graphs, created for the Neo4j graph database.
Example: Cypher query to insert left hand portion of figure:

```
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)
```

tail → (Idaho) ... head ← (USA)

Scanned with CamScanner

M T W T F S S
Page No.:
Date:
YOUVA

Example: Cypher query to 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
```

The query can be read as follows :-

Find any vertex (call it person) that meets both of the following conditions:

1. person has an outgoing BORN_IN edge to some vertex. From that vertex, you can follow a chain of outgoing WITHIN edges until eventually you reach a vertex of type Location, whose name property is equal to "United States".

2. That same person vertex also has an outgoing LIVES_IN edge. Following that edge, and then a chain of outgoing WITHIN edges, you eventually reach a vertex of type Location, whose name property is equal to "Europe".

For each such person vertex, return the name property

WITHIN*0.. → "follow a WITHIN edge, zero or more times"

Scanned with CamScanner

Graph Queries in SQL :-

* In a relational database, you usually know in advance which joins you need in your query.
* In a graph query, the number of joins is not fixed in advance.
* Variable - length traversal paths :
  - in Cypher : `() -[:WITHIN*0..]-> ()`
    LIVES_IN edge may point at any kind of location & LIVES_IN edge may point directly at location vertex you're looking for.
  - SQL: 1999 : this idea of variable - length traversal paths in a query can be expressed using something called "recursive common table expressions" (the WITH RECURSIVE syntax)

```
WITH RECURSIVE
    -- in_usa is the set of vertex IDs of all locations
       within the United States
    in_usa(vertex_id) AS (
        SELECT vertex_id FROM vertices WHERE properties ->>
                'name' = 'United States'  (1)
        UNION
        SELECT edges.tail_vertex FROM edges  (2)
            JOIN in_usa ON edges.head_vertex = in_usa.
                                               vertex_id
            WHERE edges.label = 'within'
    ),
    -- in_europe is the set of vertex IDs of all locations
       within Europe
    in_europe(vertex_id) AS (
        SELECT vertex_id FROM vertices WHERE properties ->>
                'name' = 'Europe'  (3)
```

Scanned with CamScanner

```
    UNION
        SELECT edges.tail_vertex FROM edges
            JOIN in_europe ON edges.head_vertex =
                                in_europe.vertex_id
            WHERE edges.label = 'within'
),
-- born_in_usa is the set of vertex IDs of all people
   born in the US
born_in_usa (vertex_id) AS ( (4)
    SELECT edges.tail_vertex FROM edges
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'born_in'
),
-- lives_in_europe is the set of vertex IDs of all
   people living in Europe.
lives_in_europe (vertex_id) AS ( (5)
    SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex =
                                in_europe.vertex_id
        WHERE edges.label = 'lives_in'
)

SELECT vertices.properties ->> 'name'
FROM vertices
-- join to find those people who were both born in the
    US and live in Europe.
JOIN born_in_usa ON vertices.vertex_id = born_in_usa.
                                vertex_id (6)
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.
                                vertex_id;
```

Scanned with CamScanner

1. First find the vertex whose name property has the value "United States", and make it the first element of the set of vertices in_usa

2. Follow all incoming within edges from vertices in the set in_usa, and add them to the same set, until all incoming within edges have been visited.

3. Do the same starting with the vertex whose name property has the value "Europe", and build up the set of vertices in_europe

4. For each of the vertices in the set in_usa, follow incoming born_in edges to find people who were born in some place within the United States.

5. Similarly, for each of the vertices in the set in_europe, follow incoming lives_in edges to find people who live in Europe.

6. Finally, intersect the set of people born in the USA with the set of people living in Europe, by joining them.

Scanned with CamScanner

M T W T F S S
Page No.:
Date:
YOUVA

10. Triple-stores and SPARQL :-

* mostly equivalent to the property graph model.

* In Triple-store, all information is stored in the form of very simple three-part statements :

	Eg:
* subject	* Jim
* predicate	* likes
* object	* bananas.

* The "subject" of a triple <=> "vertex" in a graph.
* The "object" is one of two things :

1. A value in a primitive datatype, such as a string or a number.
In that case, the predicate and object of the triple are equivalent to the key and value of a property on the subject vertex.
Eg: (lucy, age, 33)
vertex : lucy
properties : {age: 33}

2. Another vertex in the graph. In that case, the predicate is an edge in the graph, the subject is the tail vertex, and the object is the head vertex.
Eg: (lucy, marriedTo, alain)
vertices : lucy, alain
edge: 'marriedTo' is a label.

Scanned with CamScanner

* Previous example of graph model into Triplet Store model:-

data written as triplets in a format called "Turtle"

→ Turtle Triples

```
@prefix : <urn:example:>
_:lucy      a          :Person
_:lucy      :name      "Lucy"
_:lucy      :bornIn    _:idaho
_:idaho     a          :Location
_:idaho     :name      "Idaho"
_:idaho     :type      "state"
_:idaho     :within    _:usa
_:usa       a          :Location
_:usa       :name      "United States"
_:usa       :type      "country"
_:usa       :within    _:namerica
_:namerica  a          :Location
_:namerica  :name      "North America"
_:namerica  :type      "continent"
```

* A more concise way:-

```
@prefix : <urn:example:>
_:lucy      a :Person;   :name "Lucy"          :bornIn _:idaho.
_:idaho     a :Location; :name "Idaho"         :type "state"; :within _:usa
_:usa       a :Location; :name "United States" :type "country"; :within _:namerica
_:namerica  a :Location; :name "North America"; :type "continent".
```

Scanned with CamScanner

* The semantic web :-

The triple-store data model is completely independent of the semantic web - for example, Datomic is a triple-store that does not claim to have anything to do with it.

The semantic web is a simple and reasonable idea: websites already publish information as text and pictures for humans to read, so why don't they also publish information as machine-readable data for computers to read?

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".

The semantic web was overhyped in the early 2000s but so far hasn't shown any sign of being realized in practice.

However, there is also a lot of good work that has come out of the semantic web project. Triples can be a good internal data model for applications, even if you have no interest in publishing RDF data on the semantic web.

Scanned with CamScanner

M T W T F S S
Page No.
Date
YOUVA

* The RDF data model :-

The Turtle language in the previous query is a human-readable format for RDF data.

Sometimes RDF is also written in an XML format

Example: The data of previous example expressed using RDF/XML format

```
<rdf:RDF xmlns="urn:example:" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Location rdf:nodeID="idaho">
        <name>Idaho</name>
        <type>state</type>
        <within>
            <Location rdf:nodeID="usa">
                <name>United States</name>
                <type>country</type>
                <within>
                    <Location rdf:nodeID="namerica">
                        <name>North America</name>
                        <type>continent</type>
                    </Location>
                </within>
            </Location>
        </within>
    </Location>

    <Person rdf:nodeID="lucy">
        <name>Lucy</name>
        <bornIn rdf:nodeID="idaho"/>
    </Person>
</rdf:RDF>
```

Scanned with CamScanner

* In RDF, subject, predicate, and object of a triple are often URIs. predicate: <http://my-company.com/namespace#within> or <http://my-company.com/namespace#lives_in>, rather than just WITHIN or LIVES_IN.
* The reasoning behind this design is that you should be able to combine your data with someone else's data, and if they attach a different meaning to the word within or lives_in, you won't get a conflict because their predicates are actually <http://other.org/foo#within> and <http://other.org/foo#lives_in>.

* The SPARQL query language:-

* SPARQL is a query language for triple-stores using the RDF data model. (Acronym for SPARQL Protocol and RDF Query Language, pronounced "sparkle")
* The same query as before - finding people moved from US to Europe - is even more concise in SPARQL than it is in Cypher.

```
PREFIX : <urn:example:>
SELECT ?personName WHERE {
    ?person :name ?personName.
    ?person :bornIn / :within* / :name "United States".
    ?person :livesIn / :within* / :name "Europe".
```

variables start with ? in SPARQL

```
# Cypher (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)
# SPARQL ?person :bornIn / :within* ?location.

# Cypher (usa {name: 'United States'})
# SPARQL ?usa :name "United States".
```

Scanned with CamScanner

* The Foundation : Datalog :

Datalog is a much older language than Cypher or SPARQL, which provided foundation for later query languages

It is the query language of Datomic and Cascalog is a Datalog implementation for querying large datasets in Hadoop.

Datalog's data model is similar to the triple-store model, generalized a bit.

triple-store	(subject, predicate, object)
Datalog	predicate (subject, object)

eg:- age (lucy, 33).

The difference of Datalog from Triples is that we set rules for predicates.

Means, we predefine predicates instead of storing them as data in database.

Scanned with CamScanner

* Summary :-

* Historically, data started out being represented as one big tree (the hierarchical model) but that wasn't good for many-to-many relationships.

* So the relational model was invented to solve that problem.

* More recently, developers found that some applications don't fit well in the relational model either. New non-relational "NoSQL" datastores have diverged in two main directions:

* 1. Document databases target use cases where data comes in self-contained documents and relationships between one document and another are rare.

* 2. Graph databases go in the opposite direction, targeting use cases where anything is potentially related to everything.

* Document and graph models don't enforce a strict schema for the data they store, which can make it easier to adapt applications to changing requirements. However, your application most likely still assumes that data has a certain structure; it's just a question of whether the schema is explicit (enforced on write) or implicit (handled on read).

Scanned with CamScanner

* Each data model comes with its own query language. We discussed examples:

SQL, MapReduce, MongoDB's aggregation pipeline, Cypher, SparQL, Datalog, CSS.

* There are many other data models like

for "sequence-similarity searches", for searching DNA molecule represented by string among large database of strings

"Large Hadron Collider (LHC)" project for BigData-style large-scale data analysis working with hundreds of petabytes.

"Full-text search"

Scanned with CamScanner