Not Only SQL

Knowledge objectives

1. Define the impedance mismatch
2. Identify applications handling different kinds of data
3. Name four different kinds of data models
4. Explain three consequences of schema variability
5. Explain the consequences of physical independence
6. Explain the difference between relational and correlational models
7. Explain the relationship between arrays and 4NF
8. Compare the three possibilities to represent multivalued attributes
9. Name two implementations of semistructured data
10. Explain the design principle of documents
11. Name 3 consequences of the design principle of a document store
12. Explain the difference between relational foreign keys and document references

Understanding objectives

- Given two alternative structures of a document, explain the performance impact of the choice in a given setting

Application objectives

- Given a relatively small relational schema and some queries over it, transform it into a more efficient semi-structured schema
- Transform some SQL queries over a schema in 1NF into equivalent queries over another schema containing JSON documents
- Given a multivalued attribute, choose the best implementation in a given setting

Motivation

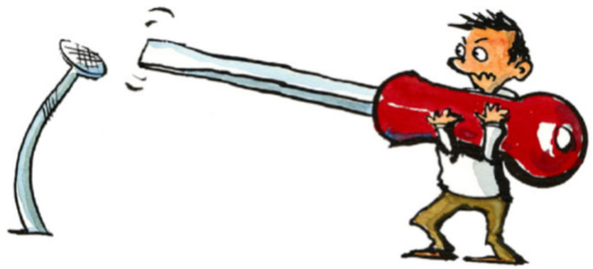
From SQL to NOSQL

Law of the instrument

"Over-reliance on a familiar tool."

Wikipedia

- *Golden hammer* anti-pattern: "A *familiar technology or concept applied obsessively to many software problems.*"



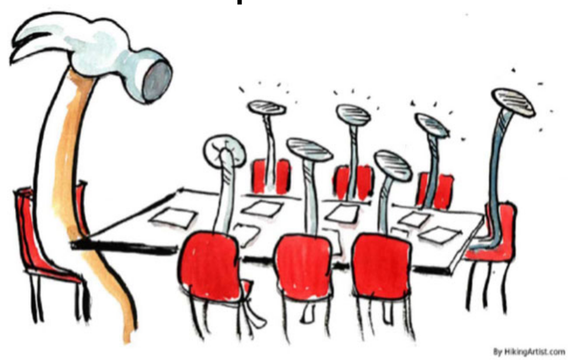
If the only tool you have is a hammer,
everything looks like a nail.

Law of the Relational Database

Object-relational impedance mismatch is "... *one in which a program written using an object-oriented language uses a relational database for storage.*"

Ireland et al.

- Since we only know relational databases, every time we want to model a new domain we'll automatically think on how to represent it as columns and rows



If the only tool you have is a relational database, everything looks like a table.

One size does not fit all

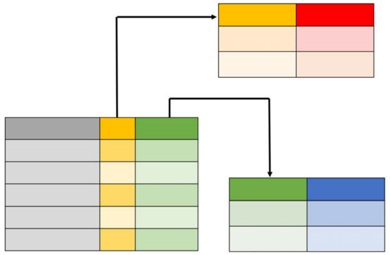
Not Only SQL (different problems entail different solutions)

- OLTP
  - VoltDB, HANA, Hekaton
- Data warehousing and OLAP
  - Vertica, Red Shift, Sybase IQ
- Scientific data
  - R, Matlab, SciDB
- Semantic Web and Open Data
  - Virtuoso, GraphDB
- Text
  - Google, Yahoo
- Documents (i.e., XML, JSON)
  - MongoDB, CouchDB
- Stream processing
  - Storm, Spark Streaming, Flink

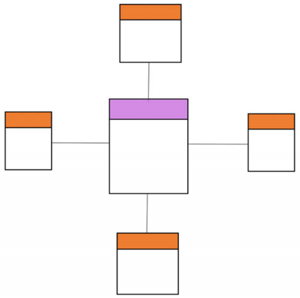
Michael Stonebraker: Technical perspective - One size fits all: an idea whose time has come and gone. Commun. ACM, 2008

Different data models

Relational



Multidimensional



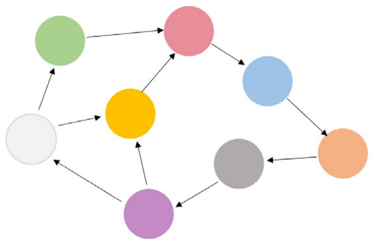
Key-Value

KEY	VALUE
KEY	VALUE
KEY	VALUE
KEY	VALUE
KEY	VALUE

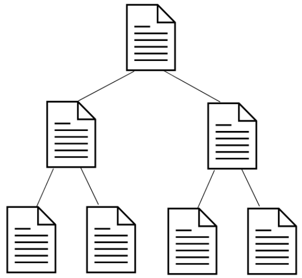
Wide-Column

	Family1	Family2	Family3	Family4
Key				
Key				
Key				
Key				

Graph



Document



By Aina Montalban, inspired by Daniel G. McCreary and Ann M. Kelly

Database models

RELATIONAL

- Based on mathematical theory
  - Sets, instances and attributes
    - Tables, rows and columns
  - Constraints are allowed
    - PK, FK, Check, …

When creating the tables you MUST specify their schema (i.e., columns and constraints)

Data is restructured when brought into memory (impedance mismatch)

NOSQL

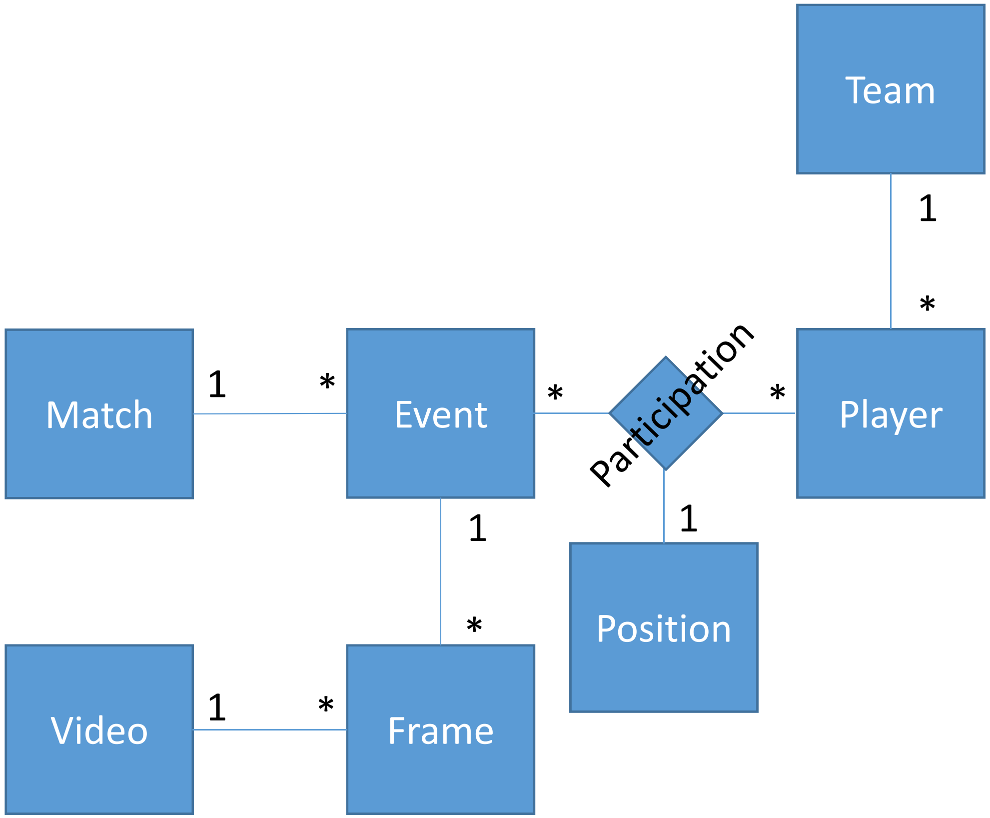
- No single reference model
  - Graph data model
  - Document-oriented databases
  - Key-value (~ hash tables)
  - Streams (~ vectors and matrixes)

Ideally, schema specified at insertion, not at definition (schemaless databases)

The closer the data model in use looks to the way data is stored internally the better (read/write through)

Schema definition

Events example



Match(<u>mID</u>, name, team, ...)

Video(<u>vID</u>, filename)

Event(<u>eID</u>, kind, duration, ..., mID)

Frame(<u>fID</u>, eID, vID)

Team(<u>tID</u>, name)

Player(<u>pID</u>, name, ..., tID)

Participation(<u>eID, pID</u>, x, y)

1NF example

Match			
mID	Name	Team	…
1	FCB-SFC	First	…

Video	
vID	CustKey
15	file://c:/...

Frame		
fID	eID	vID
8	8	15

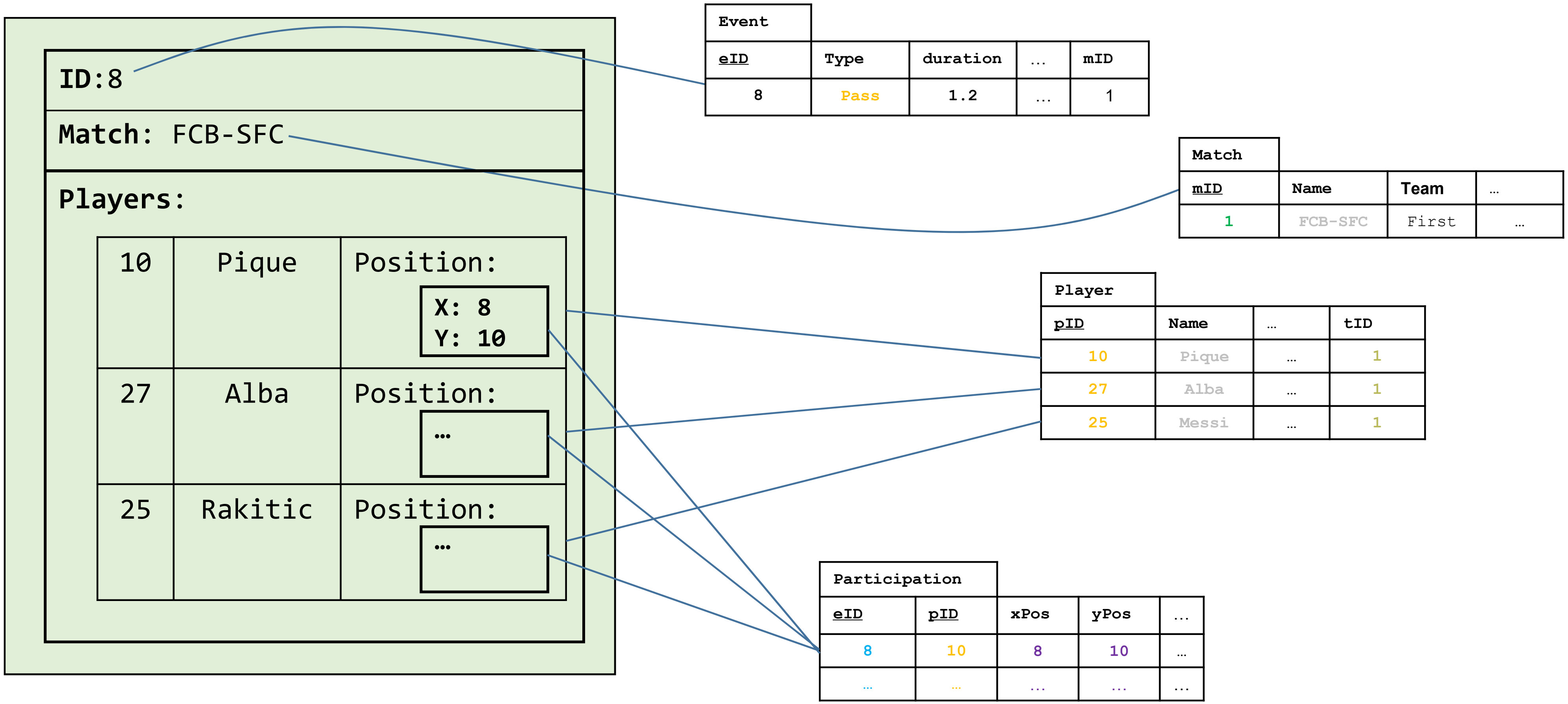
Event				
eID	Type	duration	...	mID
8	Pass	1.2	...	1

Player			
pID	Name	…	tID
10	Pique	…	1

Team	
tID	Name
1	FC Barcelona

Participation				
eID	pID	xPos	yPos	...
8	10	8	10	…

NF^2 Example (I)



An event is a single ***aggregate***

NF^2 Example (II)

```
//in event
{
 "eID" : 8,
 "match" : "FCB-SFC",
 "type" : "Pass",
 "duration" : 1.2,
 "players" : [
     {
       "pID" : 10,
       "name" : "Pique",
       "position" : {
           "xpos" : 8,
           "ypos" : 10
           }
     },  …]
  "origin_pos" : {
       "type" : "Point",
       "coordinates" : [
          8,
          10
       ]
    },
"destination_pos" : {
       "type" : "Point",
       "coordinates" : [
          23,
          15
       ]
    }
}
```

```
//in event
{
 "eID" : 8,
 "stat_pos_players" : [
   {
          "norm_pos_x" : 0.3060192,
          "norm_pos_y" : 0.492700235294118,
          "origin_player_name" : "PIQUE",
    },
    {
          "norm_pos_x" : 0.463992685714286,
          "norm_pos_y" : 0.0835062352941176,
          "origin_player_name" : "ALBA",
    },
    {
          "norm_pos_x" : 0.429419657142857,
          "norm_pos_y" : 0.420086117647059,
          "origin_player_name" : "BUSQUETS",
     },
     {
          "norm_pos_x" : 0.535141714285714,
          "norm_pos_y" : 0.494179411764706,
          "origin_player_name" : "MESSI",
     },
     {
          "norm_pos_x" : 0.116172342857143,
          "norm_pos_y" : 0.488128235294118,
          "origin_player_name" : "TER STEGEN",
      }, …]
}
```

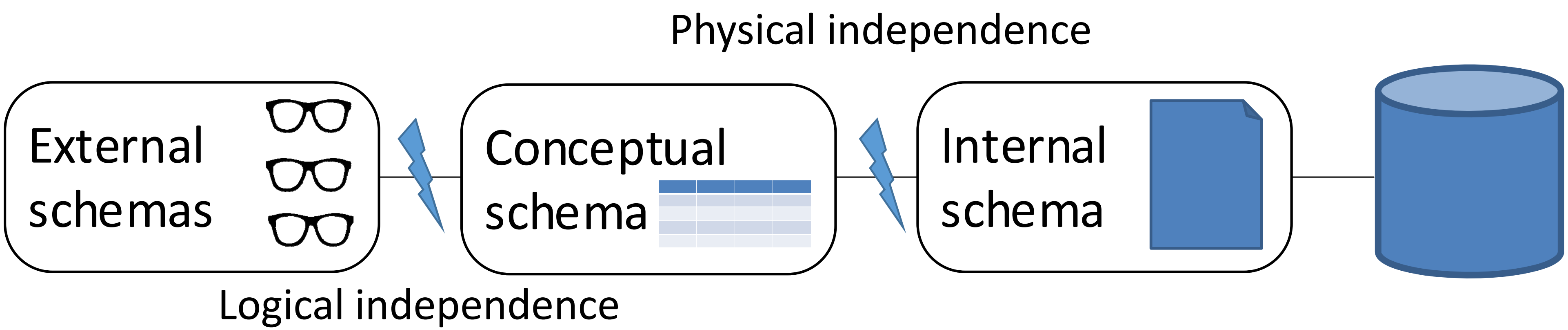
```
//in match
{
"mID" : 1,
"name" : "FCB-SFC"
…
}
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Schema variability

- `CREATE TABLE Students(id int,name varchar(50),surname varchar(50),enrolment date);`
- `INSERT INTO Students (1,'Sergi','Nadal','01/01/2012',true,'Igualada');` **WRONG**
- `INSERT INTO Students (1,'Sergi','Nadal',NULL);` **OK**
- `INSERT INTO Students (1,'Sergi','Nadal','01/01/2012');` **OK**
- Schemaless → `INSERT INTO Students (1, {'Sergi', 'Nadal', '01/01/2012', true});`
- Consequences
  - Gain flexibility
  - Lose semantics (also consistency)
  - The data independence principle is lost (!)
    - The ANSI / SPARC architecture is not followed → Implicit schema
    - Applications can access and manipulate the database internal structures

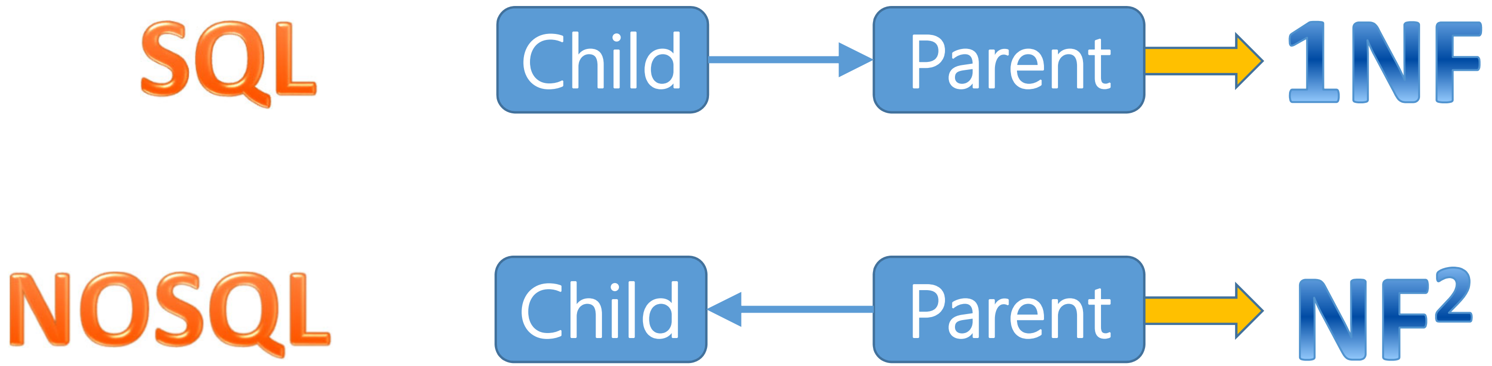
ANSI/SPARC


Physical independence
External schemas
Conceptual schema
Internal schema
Logical independence

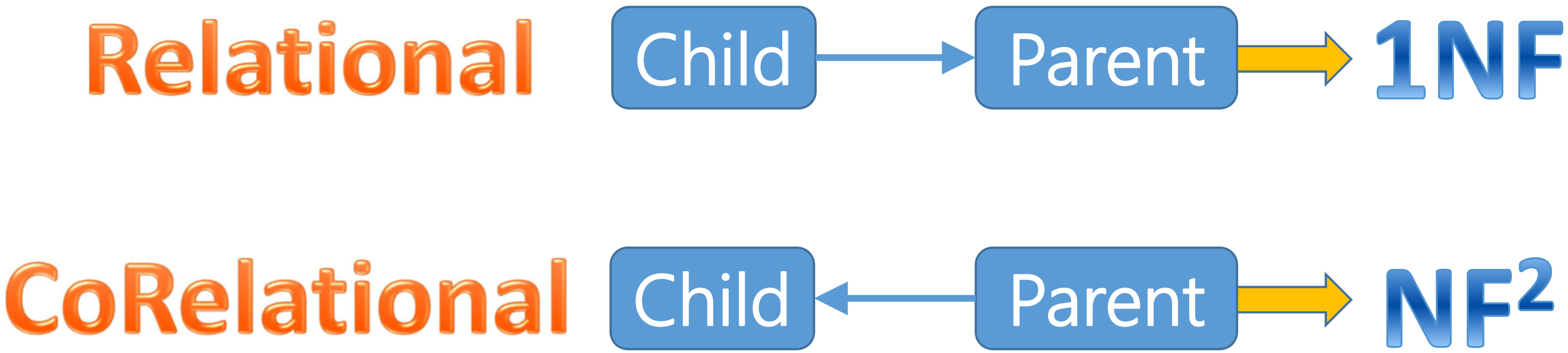
ANSI/SPARC

Storing arrays

Just Another Point of View



Just Another Point of View



Arrays in PostgreSQL

```
CREATE TABLE skills (id integer PRIMARY KEY, prog_lang TEXT ARRAY, spoken_lang TEXT ARRAY);

ALTER TABLE skills ADD CONSTRAINT new_constraint CHECK ('Spanish' <> All(spoken_lang));

INSERT INTO skills VALUES (16, '{"Java","Python"}', '{"French","English","German"}');
INSERT INTO skills VALUES (17, '{"C++","Python"}', '{"French","German"}');

UPDATE skills SET spoken_lang[1] = 'Catalan';

SELECT id, array_dims(prog_lang), array_dims(spoken_lang) FROM skills;
```

id	array_dims	array_dims
16	[1:2]	[1:3]
17	[1:2]	[1:2]

```
SELECT id, prog_lang[2], spoken_lang FROM skills;
```

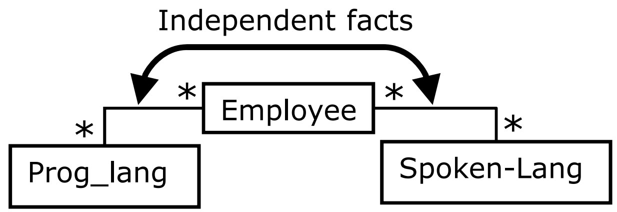
id	prog_lang	spoken_lang
16	Python	{Catalan,English,German}
17	Python	{Catalan,German}

```
SELECT * FROM skills WHERE 'English' = ANY (spoken_lang);
```

id	prog_lang	spoken_lang
16	{Java,Python}	{Catalan,English,German}

https://www.postgresql.org/docs/current/arrays.html
https://www.postgresql.org/docs/current/xaggr.html

Relationship between arrays and 4NF



(<u>id</u>, prog_lang, spoken_lang)

16	{Java,Python}	{French,English,German}
17	{C++,Python}	{French,German}

❌ 1FN? ⇒ ✔ NF^2

Flatten

```sql
SELECT id, f_prog_lang
FROM skills s CROSS JOIN UNNEST(s.prog_lang) AS f_prog_lang;
SELECT id, f_spoken_lang
FROM skills s CROSS JOIN UNNEST(s.spoken_lang) AS f_spoken_lang;
```

Denormalize

```sql
SELECT id, array_agg(f_prog_lang)
FROM table_prog
GROUP BY id;
SELECT id, array_agg(f_spoken_lang)
FROM table_spoken
GROUP BY id;
```

(<u>id, f_prog_lang</u>)

16	Java
16	Python
17	Python
17	C++

(<u>id, f_spoken_lang</u>)

16	French
16	English
16	German
17	French
17	German

✔ 1NF?

✔ 2NF?

✔ 3NF?

✔ BCNF?

✔ 4FN

Multivalued attributes comparison revisited

Per column	In an array	Per row
Fixed number of values	Variable number of values	Variable number of values
Few values	Not many values	Many values
Generates nulls	Generates empty positions	There are no null values
One I/O	One I/O	Many I/O
Global processing	Global processing	Partial processing
Natural PK	Natural PK	Artificial PK
Less space	Intermediate space	More space
Hard to aggregate	User defined aggregation	Easy to aggregate
Many CHECKs	Specific checks	One CHECK
Lower concurrency	Lower concurrency	Higher concurrency

Semi-structured database model

JSON

~~XML~~

Semi-structured data

- Document stores are essentially key-value stores
  - The value is a document
    - Allow secondary indexes
- Different implementations
  - eXtensible Markup Language (XML)
  - JavaScript Object Notation (JSON)
- Tightly related to the web
  - Easily readable by humans and machines
  - Data exchange formats for REST APIs

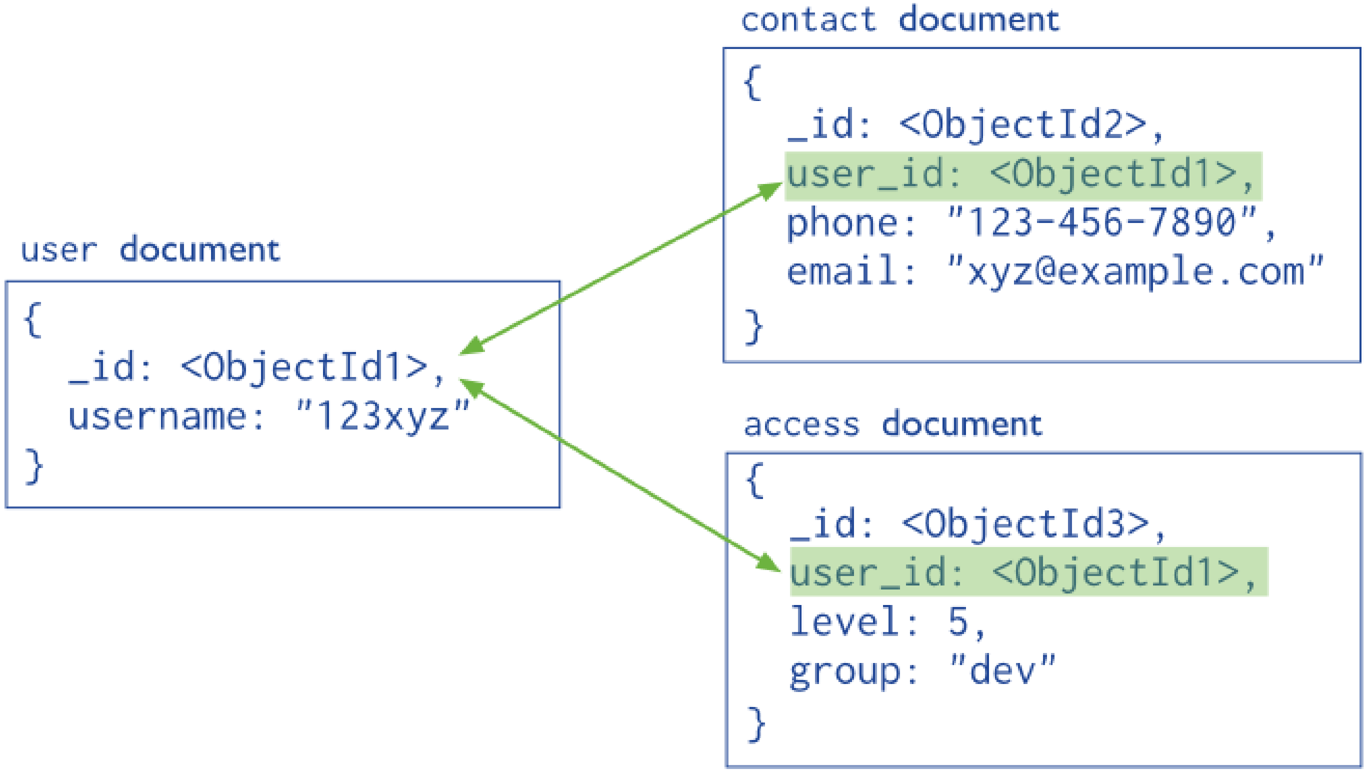
JSON Documents

- Lightweight data interchange format
- Natively compatible with JavaScript
  - Web browsers are natural clients
- Can contain unbounded nesting of arrays and objects
  - Brackets ([]) represent ordered lists
  - Curly braces ({}) represent key-value dictionaries (a.k.a. finite maps)
    - Keys must be strings, delimited by quotes (")
    - Values can be strings, numbers, booleans, lists, or key-value dictionaries
- JSON-like storage
  - *MongoDB*
  - *CouchDB*
  - Relational extensions for *Oracle*, *PostgreSQL*, etc.

JSON Example (I)

```json
{
  "title": "The Social network",
  "year": "2010",
  "genre": "drama",
  "country": "USA",
  "director": {
    "last_name": "Fincher",
    "first_name": "David",
    "birth_date": "1962"
  },
  "actors": [
    {
      "first_name": "Jesse",
      "last_name": "Eisenberg",
      "birth_date": "1983",
      "role": "Mark Zuckerberg"
    },
    {
      "first_name": "Rooney",
      "last_name": "Mara",
      "birth_date": "1985",
      "role": "Erica Albright"
    }
  ]
}
```

JSON Example (II)



JSON Example (III)

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

Designing Document Stores

Do not think relational-wise

- Break 1NF (i.e., follow NF^2) to avoid joins
  - Get all data needed with one single fetch
  - Use indexes to identify finer data granularities
- Consistency can still be checked with JSON Schema

Consequences:

- Store independent documents
  - Avoid pointers (i.e., neither FKs nor references)
    - Massive denormalization
- Massive rearrangement of documents on changing the application layout

https://json-schema.org

JSON data type

PostgreSQL

JSON vs JSONB

a) JSON
- Stores the text corresponding to the document as is (preserves formatting)
  - Keeps extra spaces between key-value pairs
  - Keeps the order of the keys
  - Keeps potentially repeated keys
    - Last one would be retrieved
- Parsing is done in every query

b) JSONB
- Stores a more efficient binary format
- Parses the document only at insertion
  - Removes spaces
  - Does not preserve the order of keys
  - Removes duplicated keys
    - Preserves the last
- Supports indexing and many different operators

https://www.postgresql.org/docs/current/datatype-json.html

JSON management in PostgreSQL: Basics

```sql
CREATE TABLE employees (dni CHAR(8), name CHAR(8), contact JSONB, PRIMARY KEY (dni) );

INSERT INTO employees VALUES ('12345678','Jordi','{"telephones":{"count":3,"fix":["934622244","934643434"],"mobile":["685481253"]}}');
INSERT INTO employees VALUES ('12345679','Anna', '{"telephones":{"count":1,"others":["934622243"],"mobile":["666666666"]}}'::jsonb);
INSERT INTO employees VALUES ('22345678','Eva',
                                jsonb_build_object('telephones', jsonb_build_object(   'count', 2,
                                                                                        'fix', jsonb_build_array('934643434'),
                                                                                        'mobile', array_to_json(ARRAY['777777777']))));


SELECT contact FROM employees WHERE dni='12345678';
          {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}

-- Access to elements in a JSON
SELECT contact['telephones']['count']::integer,
 (contact->'telephones'->'count')::integer,
 (contact#>'{"telephones","count"}')::integer
FROM employees;

     contact|int4|int4|
     -------+----+----+
           3|   3|   3|
           1|   1|   1|
           2|   2|   2|
```

https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html

JSON management in PostgreSQL: Arrays

```
dni     |name    |contact                                                                        |
--------+--------+-------------------------------------------------------------------------------+
12345678|Jordi   |{"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}|
12345679|Anna    |{"telephones": {"count": 1, "mobile": ["666666666"], "others": ["934622243"]}} |
22345678|Eva     |{"telephones": {"fix": ["934643434"], "count": 2, "mobile": ["777777777"]}}    |
```

```
SELECT (contact->'telephones'->'fix')[0],(contact->'telephones'->'fix')->0,(contact->'telephones'->'fix')->>0 FROM employees;
     ?column?   |?column?   |?column? |
     -----------+-----------+---------+
     "934622244"|"934622244"|934622244|
                |           |         |
     "934643434"|"934643434"|934643434|
```

JSON arrays are 0-relative, unlike regular SQL arrays that start from 1.

```
SELECT name, ARRAY(SELECT jsonb_array_elements_text(contact->'telephones'->'fix')) FROM employees;
     name    |array                |
     --------+---------------------+
     Jordi   |{934622244,934643434}|
     Anna    |{}                   |
     Eva     |{934643434}          |
```

```
SELECT name, value FROM employees CROSS JOIN UNNEST(ARRAY(SELECT jsonb_array_elements_text(contact->'telephones'->'fix'))) AS value;
SELECT name, value FROM employees CROSS JOIN jsonb_array_elements_text(contact->'telephones'->'fix');
     name    |value    |
     --------+---------+
     Jordi   |934622244|
     Jordi   |934643434|
     Eva     |934643434|
```

https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html

JSON management in PostgreSQL: Normalization

```
dni     |name    |contact                                                                            |
--------+--------+-----------------------------------------------------------------------------------+
12345678|Jordi   |{"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}|
12345679|Anna    |{"telephones": {"count": 1, "mobile": ["666666666"], "others": ["934622243"]}}     |
22345678|Eva     |{"telephones": {"fix": ["934643434"], "count": 2, "mobile": ["777777777"]}}        |
```

Flatten

```sql
SELECT dni, name,
  (contact->'telephones'->'count')::integer AS tel_count,
  (contact->'telephones'->'fix'->>0) AS tel_fix0,
  (contact->'telephones'->'fix'->>1) AS tel_fix1,
  (contact->'telephones'->'mobile'->>0) AS tel_mobile,
  (contact->'telephones'->'others'->>0) AS tel_others
FROM employees;
```

Denormalize

```sql
SELECT dni, name,
  jsonb_strip_nulls(jsonb_build_object('telephones', jsonb_build_object('count', tel_count,
      'fix', array_to_json(nullif(array_remove(array[tel_fix0, tel_fix1], NULL),'{}'::text[])),
      'mobile', array_to_json(nullif(array_remove(array[tel_mobile], NULL),'{}'::text[])),
      'others', array_to_json(nullif(array_remove(array[tel_others], NULL),'{}'::text[]))))
    ) AS contact
FROM employees;
```

```
dni     |name    |tel_count|tel_fix0 |tel_fix1 |tel_mobile|tel_others|
--------+--------+---------+---------+---------+----------+----------+
12345678|Jordi   |        3|934622244|934643434|685481253 |          |
12345679|Anna    |        1|         |         |666666666 |934622243 |
22345678|Eva     |        2|934643434|         |777777777 |          |
```

https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html

JSON management in PostgreSQL: Conditions

```
dni      |name     |contact                                                                            |
--------+--------+-----------------------------------------------------------------------------------+
12345678|Jordi    |{"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}|
12345679|Anna     |{"telephones": {"count": 1, "mobile": ["666666666"], "others": ["934622243"]}}         |
22345678|Eva      |{"telephones": {"fix": ["934643434"], "count": 2, "mobile": ["777777777"]}}            |
```

```sql
-- Checking the existence of a field
SELECT name FROM employees WHERE contact->'telephones' ? 'mobile';                     => Jordi, Anna i Eva

SELECT name FROM employees WHERE contact->'telephones' ?& ARRAY['fix','mobile'];       => Jordi i Eva

SELECT name FROM employees WHERE contact->'telephones' ?| ARRAY['fix','others'];       => Jordi, Anna i Eva

-- Checking document containment
SELECT name FROM employees WHERE contact @> '{"telephones": {"count":1}}';             => Anna

SELECT e1.name AS emp1, e2.name AS emp2
FROM employees e1, employees e2
WHERE e1.dni<>e2.dni
        AND (e1.contact->'telephones'->'fix') @> (e2.contact->'telephones'->'fix');    => Jordi-Eva
```

https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html

JSON management in PostgreSQL: Updates

```
SELECT contact FROM employees WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}
UPDATE employees SET contact = contact||('{"newatt": "newVal"}')::jsonb WHERE dni='12345678';
        {"newatt": "newVal", "telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}
UPDATE employees SET contact = contact||('{"newatt": "'||dni||'"}')::jsonb WHERE dni='12345678';
        {"newatt": "12345678", "telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}
UPDATE employees SET contact = contact-'newatt' WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}
UPDATE employees SET contact['telephones']['newatt'] = null WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"], "newatt": null}}
UPDATE employees SET contact['telephones']['newatt'] = '"newVal"' WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"], "newatt": "newVal"}}
UPDATE employees SET contact['telephones'] = contact['telephones']-'newatt' WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253"]}}
UPDATE employees SET contact['telephones']['mobile'][2] = '"last"' WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253", null, "last"]}}
SELECT ('["a","b","c","d"]')::jsonb-2;
        ["a", "b", "d"]
UPDATE employees SET contact['telephones']['mobile'] = contact['telephones']['mobile']-1 WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253", "last"]}}
UPDATE employees SET contact['telephones']['count'] = '4' WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 4, "mobile": ["685481253", "last"]}}
UPDATE employees SET contact=jsonb_set(contact, '{telephones,count}','3') WHERE dni='12345678';
        {"telephones": {"fix": ["934622244", "934643434"], "count": 3, "mobile": ["685481253", "last"]}}
```

https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html

Closing

Summary

- NOSQL systems
- Schemaless databases
- Impedance mismatch
- Semi-structured database model
- Relational extensions
  - Arrays
  - JSON data type

Bibliography

- C. Ireland et al. *A classification of object-relational impedance mismatch* . DBKDA 2009
- M. Stonebraker et al. *The End of an Architectural Era (It's Time for a Complete Rewrite)*. VLDB, 2007
- L. Liu, M.T. Özsu (Eds.). *Encyclopedia of Database Systems*. Springer, 2009
- R. Cattell. *Scalable SQL and NoSQL Data Stores*. SIGMOD Record 39(4), 2010
- M. Stonebraker. *SQL Databases vs. NoSQL Databases*. Communications of the ACM, 53(4), 2010
- E. Meijer and G. Bierman. *A Co-Relational model of data for large shared data banks*. Communications of the ACM 54(4), 2011
- S. Abiteboul et al. *Web Data Management*. Cambridge University Press, 2012
- P. Sadagale and M. Fowler. *NoSQL distilled*. Addison-Wesley, 2013
- V. Herrero et al. *NOSQL Design for Analytical Workloads: Variability Matters*. ER, 2016
- M. Hewasinghage et al. *On the Performance Impact of Using JSON, Beyond Impedance Mismatch*. ADBIS 2020