databases, data engineering & big data

SQL, set up env and practical work

ESME SUDRIA

Luc Marchand - luc.marchand.pro@proton.me
Maxence Talon - maxencetallon@gmail.com

Global Syllabus

01 Introduction and main concepts

02 **SQL, set up env and practical work**

03 NoSQL world

04 Introduction to Big Data & Data Engineering

05 Kafka & event driven architectures

06 Spark & Delta

07 Warehouse, DBT & BI

08 IA - MLOps & RAG

Course syllabus

01 Programming Environment

02 Structured Query Language (SQL)

03 Normal forms

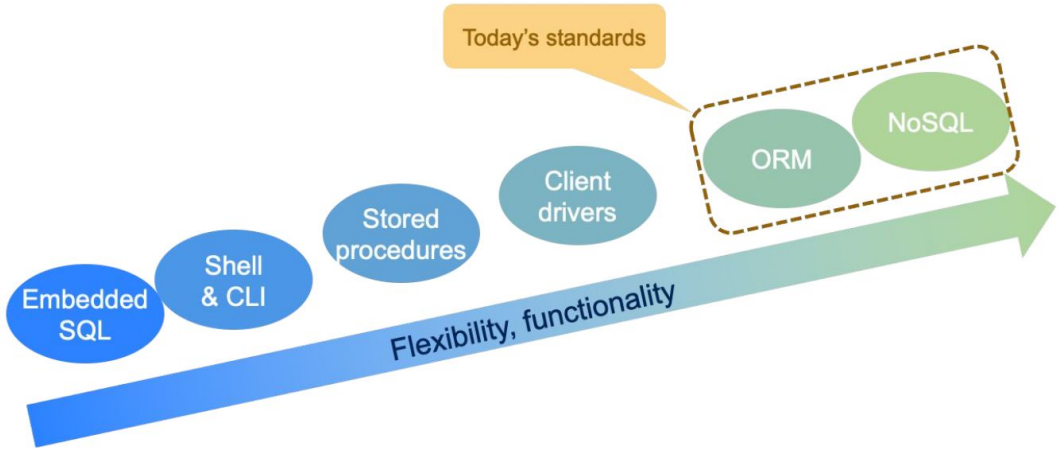
04 Modeling Patterns

01

Programming Environment

Overview

- Programming environments, in this context, are the various ways and paradigms with which a client application can interact with a DBMS
- They have been evolving regularly for the last decades, giving way to more and more flexibility and functionality



Embedded SQL

- With the Embedded SQL pattern, a **macro language** lets one intersperse SQL statements within a regular program (in C, in Java, ...)
- A pre-processing step, before compilation, translates those the macros to accesses to the database
- This was encountered in the early days of database programming, but is now superseded by the other approaches
- So, that's all for embedded SQL

Shell & command-line interface (CLI)

- These are the tools made available by DBMS vendors or third-party organizations, to run queries against interactively against a database

- Graphical tools can be used to
  - Explore the structure and the contents of a database
  - Try and refine queries before implementing them for good in an application (trial & error)
  - Administer the database
  - Examples: Oracle SQL Developer (generic), SQuirreL SQL Client (generic), Microsoft SQL Server, Management Studio (proprietary)

- Command-line tools can also meet this purpose; in addition they are used to
  - Run pre-made complex scripts, for example data migration scripts
  - Perform regular operations on a database, when run from a scheduler
  - Examples: psql, mysql shell, mysqldump, mongodb shell, ...

Stored procedures & triggers

- **Stored procedures** are functions written in a **superset** of SQL, that execute directly **on the DBMS** where the data resides
  - They appear as objects in the catalog of the DBMS

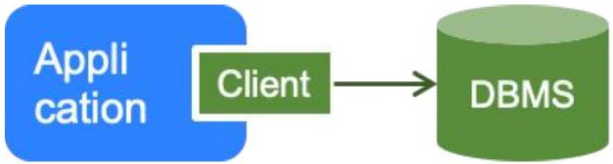
- **Triggers** are special procedures that are run by the DBMS upon data modification
  - Example with audit trail: every time a customer's address is modified, log the modification into a table

- Stored procedures and triggers feature constructs belonging to imperative programming (local variables, branches, loops, ...), which are absent from standard SQL
  - The language is usually proprietary to the DBMS vendor – a standard (SQL3) was once proposed but was never adopted

- When to use them? Usually it's best to have a**ll the business logic in one place**, i.e. in the application code (Java, C#, ...), not in the database
  - **Stored procedures** are useful when they bring a significant performance boost (there is virtually no network traffic involved since the code runs on the DBMS)
  - **Triggers** are useful when an action must absolutely be triggered, whatever the means of modification, e.g. in strong regulatory contexts. Application code cannot catch direct modifications by a rogue DBA tampering with the data directly, a trigger can

Client drivers

- Client drivers are APIs used by application code to talk to a DBMS, usually over the network. They let developers write code that emits queries, browses the results, queries the catalog, ...
  - Queries are dynamic, they can be constructed piece by piece by the program
- All client drivers revolve around the concepts of Connection, Statement and ResultSet (also called Cursor)
- There are two types of drivers
  - Native clients, provided by the DBMS vendor, which expose their own API with possible specific extensions, in the target language
  - Middlewares such as JDBC, ODBC or ADO.NET to name a few, which expose a standard API and delegate work to a proprietary client hidden to the developer
    - The abstraction from the DBMS is useful in some contexts: it allows one to have different DBMSes for development and production, for example
    - This option is almost only valid for relational DBMSes



Object - Relational Mapping (ORM)

- Object-Relational Mapping is a technique that **bridges the gap** between object-oriented programming and the relational model
- More specifically, it adds metadata to the class definitions; the metadata tells the ORM layer:
  - What table and what fields in it persist instances of a given class (i.e. entities)
  - What integrity constraints must be enforced by the database
  - How relationships between classes must be handled (foreign key reference, association table, inheritance pattern, ...)
  - Where transaction boundaries are with respect to business logic methods (aka services)
- ORM also offer several variations on query languages, for easy retrieving and updating of objects
- There is no global standard, but in the Java world the ORMs comply to the JPA (Java Persistence API) standards, also called EJB3
  - The most common JPA implementation is Hibernate
  - JPA is not limited to relational databases and thus to ORMs
  - JS: Sequelize / Python: SQLalchemy

What more ? NoSQL

- NoSQL (“Not Only SQL”), as we will see in the next module, is a different world altogether

- NoSQL DBMSes, just like relational ones, come with their specific client drivers, and sometimes with more generic API adaptors like JPA
  - For some NoSQL databases, object-to-object mapping is trivial

- While proprietary APIs are sometimes limited by the modelization and capabilities (a non transactional key/value store offers no more that get() and put() methods), the flexibility comes from the broad choice of modelization and architectures that we get
  - Until recently, the approach was: Given a RDBMS imposed upon us, what API will be most productive for our project?
  - Today, the approach is: Given our technical and functional requirements, what type of database (or combination thereof) is most tailored to suit the needs?

02

Structured Query Language (SQL)

History

- SQL (Structured Query Language) is language meant to implement tasks that are specific to relational databases
  - Some NoSQL DBMSes offer an SQL abstraction over their modelization
- It is actually a legacy of standards that, in theory, allow to change DBMS without changing the code that accesses the database

SQL-86 (ANSI) or 87 (ISO)	First standard
SQL-89	Minor adjustments
SQL-92	Major adjustments
SQL:1999 or SQL3	Adds regexps, triggers, recursivity
SQL:2003 and beyond	Add sequences, identifiers, XML, TRUNCATE, …

- In practice, most DBMSes more or less conform to the SQL-92 standard; more advanced features are usually present but with a proprietary syntax, inherited from what was once extensions to the norm
  - Other significant differences lie in data type names, function names, and syntax for procedural SQL (stored procedures and triggers)
  - Porting existing code from one DBMS to another is difficult (a bit easier with an ORM)
- The rest of this document will be written with the Postgresql syntax and dialect

Data Types

Cf postgres documentation

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

3-state logic

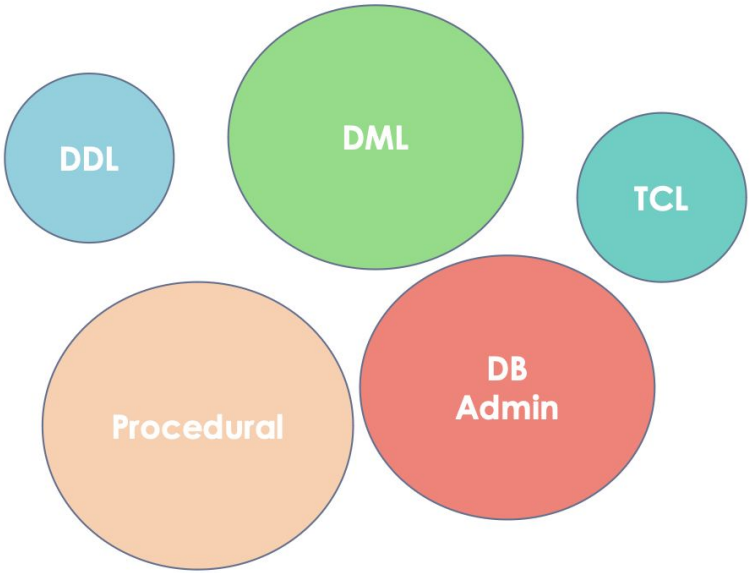
- Predicates (boolean formulas) are omnipresent in SQL, in WHERE clauses and on join conditions
  - Example: amount > 100 AND client.client_id = order.client_id
- Care must be taken when values involved in a predicate can take the NULL value: a condition involving NULL evaluates to the special boolean value UNKNOWN, and any combination of predicates involving UNKNOWN evaluates to UNKNOWN as well. UNKNOWN is neither true nor false!
- Example

order_id	order_date	customer_id
123	*NULL*	2567
124	2013-07-08	2567

  - select * from order where order_date = ’2013-07-08’ gives order no. 124, as expected
  - select * from order where order_date <> ’2013-07-08’ gives nothing!

Several languages in one

SQL – and its variants – is actually an aggregate of several languages in interaction with each other



- **DDL**: Data Definition Language
  - > Management of the objects in the database: creation, modification and deletion of tables, columns, user types, indices, procedures, …
- **DM**L: Data Manipulation Language
  - > Management of tuples: CRUD operations and more complex queries
- **TCL**: Transaction Control Language
  - > Management of transactions (open, commit, rollback)
- **Procedural**
  - > Language for imperative programming in stored procedures and triggers
- **DB Admin**
  - > Administration commands: configuration, backups, process management, …

02.1

Cheatsheet SQL

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the `country` table:

```
SELECT *
FROM country;
```

Fetch `id` and `name` columns from the `city` table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the `rating` column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the `rating` column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

#DML

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

FILTERING THE OUTPUT

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
  OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

FILTERING THE OUTPUT

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

JOIN (or explicitly **INNER JOIN**) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	**name**	**country_id**	**id**	**name**
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

QUERYING MULTIPLE TABLES

LEFT JOIN returns all rows from the left table with corresponding rows from the right table. If there's no matching row, **NULL**s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

QUERYING MULTIPLE TABLES

RIGHT JOIN returns all rows from the right table with corresponding rows from the left table. If there's no matching row, **NULL**s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

QUERYING MULTIPLE TABLES

FULL JOIN (or explicitly **FULL OUTER JOIN**) returns all rows from both tables – if there's no matching row in the second table, **NULL**s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	**name**	**country_id**	**id**	**name**
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

QUERYING MULTIPLE TABLES

CROSS JOIN returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;

SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

QUERYING MULTIPLE TABLES

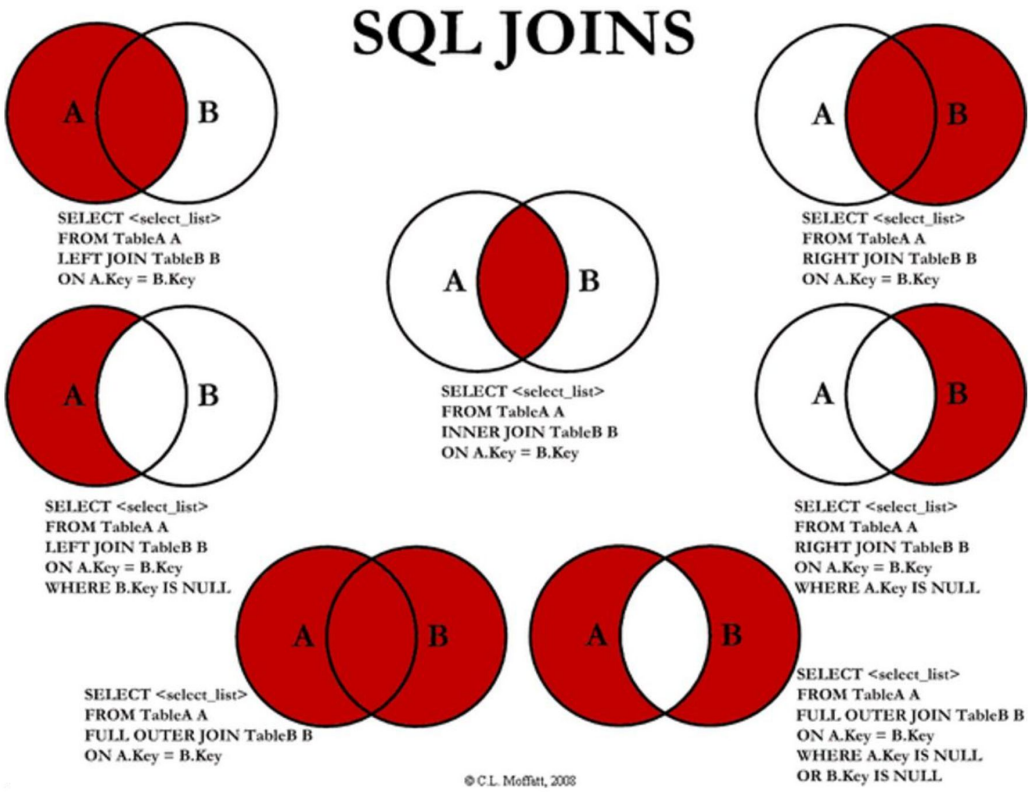
NATURAL JOIN will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

NATURAL JOIN used these columns to match rows:
city.id, **city.name**, **country.id**, **country.name**.
NATURAL JOIN is very rarely used in practice.

CHEATSHEET JOIN



AGGREGATION AND GROUPING

GROUP BY **groups** together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.

CITY		
id	**name**	**country_id**
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

CITY	
country_id	**count**
1	3
2	3
4	2

AGGREGATION AND GROUPING

AGGREGATE FUNCTIONS

- `avg(expr)` – average value for rows within the group
- `count(expr)` – count of values for rows within the group
- `max(expr)` – maximum value within the group
- `min(expr)` – minimum value within the group
- `sum(expr)` – sum of values within the group

AGGREGATION AND GROUPING

EXAMPLE QUERIES

Find out the number of cities:

```
SELECT COUNT(*)
FROM city;
```

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

AGGREGATION AND GROUPING

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.
This query finds cities with the same rating as Paris:

```
SELECT name
FROM city
WHERE rating = (
  SELECT rating
  FROM city
  WHERE name = 'Paris'
);
```

SUBQUERIES

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators `IN`, `EXISTS`, `ALL`, or `ANY`.
This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
  SELECT country_id
  FROM country
  WHERE population > 20000000
);
```

SUBQUERIES

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
  SELECT AVG(population)
  FROM city average_city
  WHERE average_city.country_id = main_city.country_id
);
```

SUBQUERIES

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
  SELECT *
  FROM city
  WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING

id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING

id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

SET OPERATIONS

UNION

UNION combines the results of two result sets and removes duplicates. **UNION ALL** doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

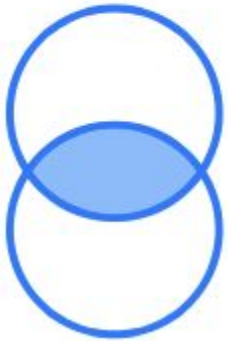
SET OPERATIONS

INTERSECT

INTERSECT returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```



SET OPERATIONS

EXCEPT

EXCEPT returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```

Data Manipulation Language (DML) Commands

Command	Description	Syntax	Example
SELECT	The SELECT command retrieves data from a database.	`SELECT column1, column2 FROM table_name;`	`SELECT first_name, last_name FROM customers;`
INSERT	The INSERT command adds new records to a table.	`INSERT INTO table_name (column1, column2) VALUES (value1, value2);`	`INSERT INTO customers (first_name, last_name) VALUES ('Mary', 'Doe');`
UPDATE	The UPDATE command is used to modify existing records in a table.	`UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;`	`UPDATE employees SET employee_name = ‘John Doe’, department = ‘Marketing’;`
DELETE	The DELETE command removes records from a table.	`DELETE FROM table_name WHERE condition;`	`DELETE FROM employees WHERE employee_name = ‘John Doe’;`

Data Definition Language (DDL) Commands

Command	Description	Syntax	Example
CREATE	The CREATE command creates a new database and objects, such as a table, index, view, or stored procedure.	`CREATE TABLE table_name (column1 datatype1, column2 datatype2, ...);`	`CREATE TABLE employees ( employee_id INT PRIMARY KEY, first_name VARCHAR(50), last_name VARCHAR(50), age INT );`
ALTER	The ALTER command adds, deletes, or modifies columns in an existing table.	`ALTER TABLE table_name ADD column_name datatype;`	`ALTER TABLE customers ADD email VARCHAR(100);`
DROP	The DROP command is used to drop an existing table in a database.	`DROP TABLE table_name;`	`DROP TABLE customers;`
TRUNCATE	The TRUNCATE command is used to delete the data inside a table, but not the table itself.	`TRUNCATE TABLE table_name;`	`TRUNCATE TABLE customers;`

OPTIMISATION

- When you want to optimize your queries, **YOU SHOULD ALWAYS MEASURE BEFORE OPTIMIZE !!**

- To measure, you can use the EXPLAIN COMMAND. This command is place before anything else in the query
  - Example : EXPLAIN SELECT * FROM city; #ADD EXAMPLE

- To optimize your query reading, you can use INDEX.
  - I let you play with generative AI and internet to figure out more. Don’t hesitate to share what you’ve found with me.

03

Normal forms

What are normal forms ?

- **Normal forms** are groups of rules that assess how "good" a database model is
  - A "good" model is a model that features no redundancy and whose very design limits the risk of data inconsistency to a minimum

- There exists plenty of normal forms (at least 1NF, 2NF, ..., 6NF, EKNF, BCNF, DKNF), but in practice the first 3 are considered
  - The first 3 normal forms are noted 1NF, 2NF and 3NF; they apply to individual relations. We say a data model is in 3NF if all its relations are in 3NF
  - They are refinements of each other: 3NF compliance implies 2NF compliance, which in turn implies 1NF
  - A model as a whole can be a mix of several normal forms

- As database designers, **we want to achieve 3NF in all our PDMs**

Normal forms apply to the way a model is **designed**. Assessing the data stored in a database is **not enough** to claim that the model is in xNF – some flaws may be hidden because the data we see doesn't show corner cases.

1NF AKA 1st Normal Form

- A relation is in 1NF if the domain of all its attributes is atomic, i.e. is not repetitive and not compound

customer_id	name	customer_address
124	John Doe	[34 Horseradish Ave @ NY]
125	Mary Smith	[12 Maniac Bvd @ Seattle; 1 Torvalds Avenue @ Boston; 384 Bull Bvd @ NY]

Blatant work around the lack of collections in the Relational Model. Risk of inconsistencies if a bug writes a string in a different format.

A step towards normalization

customer_id	name	addr_no	customer_address	city
124	John Doe	1	34 Horseradish Ave	NY
125	Mary Smith	1	12 Maniac Bvd	Seattle
125	Mary Smith	2	1 Torvalds Avenue	Boston
125	Mary Smith	3	384 Bull Bvd	NY

(We could go further by splitting customer_address into finer components)

2NF AKA 2nd Normal Form

- A relation is in 2NF if it is in 1NF and if no non-key attribute depends on a subset of a key

customer_id	name	addr_no	customer_address	city
124	John Doe	1	34 Horseradish Ave	NY
125	Mary Smith	1	12 Maniac Bvd	Seattle
125	Mary Smith	2	1 Torvalds Avenue	Boston
125	Mary Smith	3	384 Bull Bvd	NY

Here `name` depends on `customer_id`, which is a subset of the key (`customer_id`, `customer_address`). There is a risk of having different names for the same `customer_id`

A step towards normalization

2NF

customer_id	name
124	John Doe
125	Mary Smith

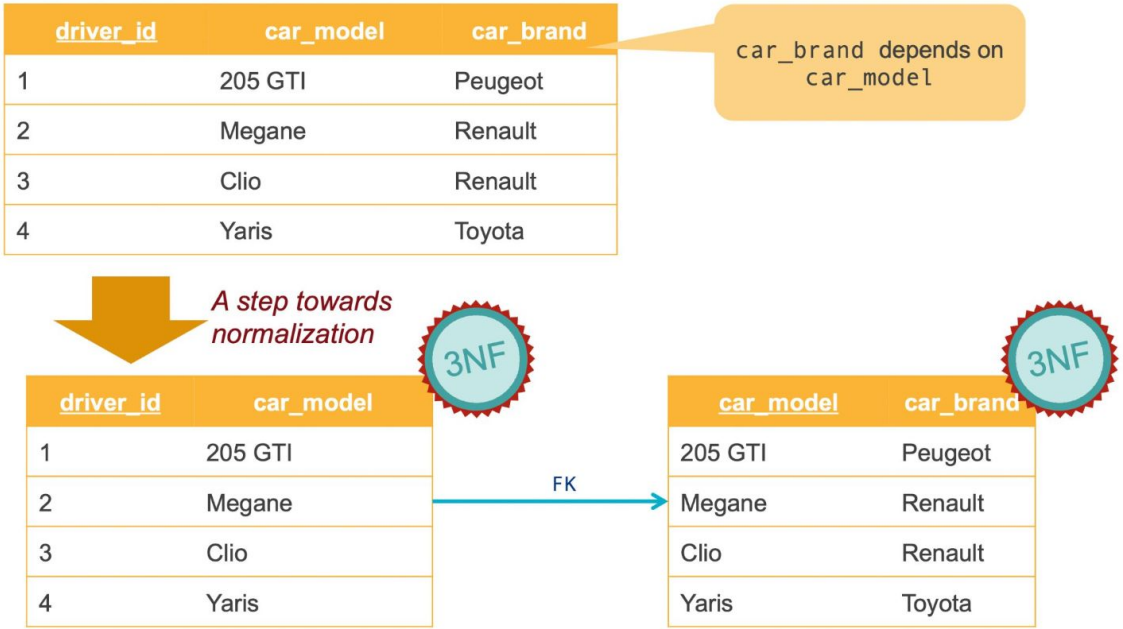
FK

2NF

customer_id	addr_no	customer_address	city
124	1	34 Horseradish Ave	NY
125	1	12 Maniac Bvd	Seattle
125	2	1 Torvalds Avenue	Boston
125	3	384 Bull Bvd	NY

3NF AKA 3rd Normal Form

- A relation is in 3NF if it is in 2NF and if no non-key attribute depends on another non-key attribute



04

Modeling patterns

More patterns

- The following slides describe useful patterns that are frequently encountered. Some are valid only in the context of a relational data model, others can be used in virtually any context

- **Data governance**: related to the life cycle of data
  - Record history (aka versioning)
  - Audit trail
  - Soft delete
  - Evolutionary Database Design

- **Development**: technical patterns that implement useful functionality
  - Key/value stores
  - Application lock
  - Generalisation

- **Data warehousing & BI**: designed specifically for OLAP workloads
  - Star, snowflake and constellation schemas
  - Denormalization

Data Governance

Record history (aka Versioning)

- Problem
  - We want to be able to "go back in time" and fetch an old version of a record from a table .i.e. we want to keep an history of records as they are modified by users

- Solution
  - The primary key of the relation is augmented with a monotonous attribute, the **version number** of the record (it may be unique for a given entity, or for all entities)
  - Two attributes give the **validity dates** of a record(start and end date/times). A NULL end date means this is the current version (maximum version number)
  - To go back in time, we just add a criterion to our queries: the target date must be between the start and end dates of the entity we are querying

- Gotchas
  - When a group of entities must be historized together, don't forget the association tables (N-M relations)!

customer_id	version	name	customer_address	valid_from	valid_to
124	1	John Doe	34 Horseradish Ave	2010-01-01	*NULL*
125	1	Mary Smith	12 Maniac Bvd	2013-02-03	2013-03-08
125	2	Mary Smith	1 Torvalds Avenue	2013-03-08	2013-09-18
125	3	Mary Smith	384 Bull Bvd	2013-09-18	*NULL*

Data Governance

Audit trail (very, very common)

- Problem
  - The application lets user modify data at will, and we want to keep track of who made what (to prevent fraud, or in case we are audited by an institution)
- Solution
  - Several attributes are added to the table; typically they tell **who created** the record and **when**, and **who modified** it for the last time and **when** (a NULL modification user & date means the record was never touched)
  - If more precision is required, we can also create a specific table aside, storing the history of all modifications, like a **log**
- Gotchas
  - Of course that is not enough to grasp the full behaviour of our users, application logs are also necessary

cust_id	name	created_by	created_on	last_update_by	last_update_on
124	John Doe	user1234	2010-01-01	*NULL*	*NULL*
125	Mary Smith	user9877	2013-02-03	user9877	2013-04-01
125	Mary Smith	user1039	2013-03-08	user1234	2013-09-28
125	Mary Smith	user1234	2013-09-18	user9877	2013-09-19

... or ...

table_name	field_name	old_value	new_value	who	when
customer	name	John Dof	John Doe	user1234	2013-09-19
user	last_name	Dof	Doe	user1234	2013-09-19
product	provider_id	1234	713	user9877	2013-09-28

Data Governance

Soft delete

- Problem
  - Sometimes, we don't want to delete a record forever from the database. Maybe by law we are not allowed to do so, or maybe it holds important information that must not disappear into the void, maybe we want to be able to undo the deletion later…
  - Example: a table with users. User logins appear in many other tables following our super-useful audit trail pattern, but the login information alone is not enough: we must keep all the information related to every use, in the USERS table
- Solution
  - An attribute is added to the table; it's a **flag** telling whether the record was "deleted" (actually it
  - never gets deleted)
  - Queries that don't want to see the "deleted" records add a criterion on that flag
  - If the table is already historized (see above), we can also use the end date as a flag(remember, it's NULL for the current record. If there is no record with a NULL end date, then it's not valid anymore)

customer_id	name	customer_address	is_deleted
124	John Doe	34 Horseradish Ave	N
125	Abdel Eted	12 Maniac Bvd	Y
126	Mary Smith	1 Torvalds Avenue	N

Data Governance

Evolutionary database design

- Problem
  - > We want to trace our database schema evolution
  - > We want to automate our schema modification in our delivery process
- Solution
  - > Add a table schema version with record for every changeset before their application
  - > Check before you apply your changeset :
    - + a record with your script name does not exists
    - + a record is not marked as failure
  - > Apply your change in a transaction
  - > Set the success flag at True, and commit the transaction

version	script	Install by	Installed on	success
124	T20150925_1800__create_table.sql	user	2016-10-03	TRUE
125	T20150925_1900__import.sql	user	2016-10-03	TRUE
126	…			

Tools as FlywayDb, Liquibase, DoctrineMigration, Active Record Migration implements this pattern and give access to command to manage your Database Lifecycle in your software factory

Development

Key / values stores

- Problem
  - We have objects that are too versatile to fit in the rigidity of the Relational model. Stil lwe need to use a relational DBMS (RDBMS)
- Solution
  - Simulate a NoSQL store, by designing a table that has only 2 attributes: a **key** and an **opaque value** (a long character string or a BLOB)
  - Note that the table is not even in1NF…
- Gotchas
  - It's difficult (and not efficient) to query the data with a criterion that refers to the contents of the BLOB. For this purpose we'd rather use a NoSQL store (but sometimes we don't have a choice ☹)
  - Similarly, updates to a particular sub-field inside a BLOB require fetching and updating the whole BLOB at once, this is coarse and not efficient when the objects are big

product_id	contents
124	<product><name>Computer</name><features><cpu>…</features></product>
125	<product><name>Blender</name><features><rpm>…</features></product>

Development

Application Lock

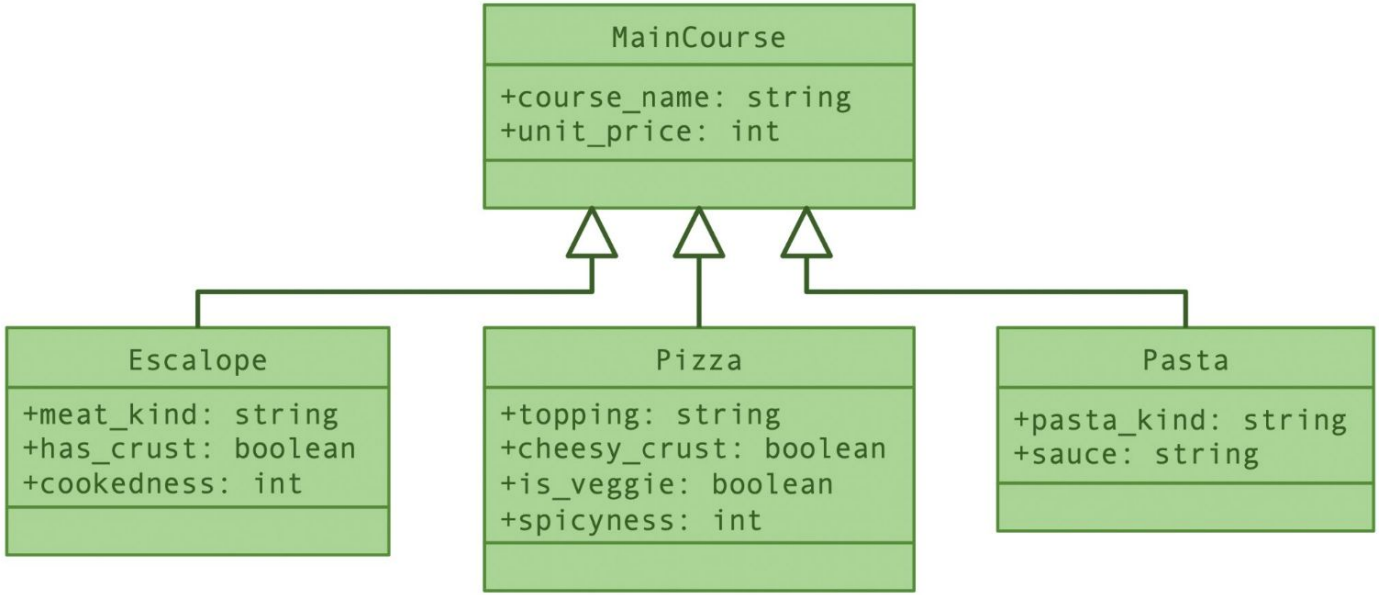
- Problem
  - > We want to prevent 2 users from modifying the same record at the same time, otherwise they will overwrite each other's modifications. More specifically, we want to notify users when they are trying to modify a record that someone else is already modifying, displaying the name of the other person
    - \+ DBMS transactions don't allow us to do that, they're too technical. Besides, we don't want to mess with transaction boundaries
- Solution
  - > Add a couple of attributes to the table, stating **who** opened (locked) the record and **when**
  - > The locking date and time is necessary because sometimes users will forget to close a screen; or the application will crash, leaving the lock set. With the date and time of locking we can set an **expiration policy** on the lock
  - > When the modification is done, release the lock by setting those fields to NULL

customer_id	name	customer_address	locked_by	locked_on
124	John Doe	34 Horseradish Ave	user1234	14:31:02
125	Abdel Eted	12 Maniac Bvd	user9877	14:54:03
126	Mary Smith	1 Torvalds Avenue	*NULL*	*NULL*

Development

Generalization

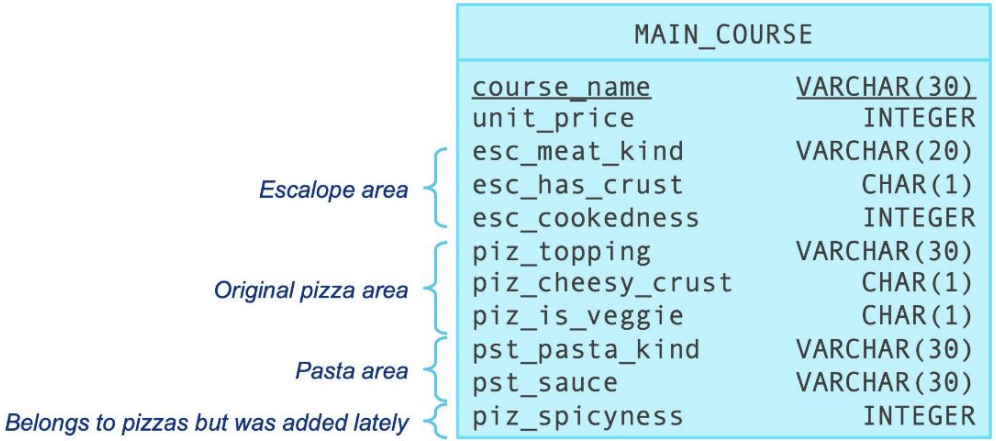
- **Generalization** is a concept specific to object-oriented programming, and thus in our case to the UML class diagram
- There is no such concept in the relational model but we can simulate it



Development

Generalization

- **Generalization pattern #1**: one single relation, with NULLable attributes corresponding to all possible subclasses
  - > Cumbersome when there is a large number of subclasses or when they have many attributes
  - > Changing one subclasse or adding a new one changes all the schema (and it's hard to keep columns of a table grouped together, new ones are added to the end!)
  - > Leads to lots of NULL values in actual data because objects are instances of a single class. Attributes corresponding to the same subclass must be all NULL or non-NULL: risk on data quality



Development

Generalization

- **Generalization pattern #2**: one relation per subclass (+1 for the parent class if it's not abstract), with the parent's attributes repeated in each relation
  - > Cumbersome when there is a large number of common attributes, changing the parent class implies modifying the schema of all subclasses. But each subclass can change without affecting the others
  - > No implicit (and error-prone) rule involving NULL values to discriminate records between subclasses
  - > The parent class's identifier should be unique across all child relations, but the Relational model can't express such a constraint involving multiple relations
  - > Searching for an object without knowing its concrete type in advance implies looking up every subclass relation, one by one
  - > The parent-child class relationships is not explicit in the PDM – all relations are totally independent

ESCALOPE	
course_name	VARCHAR(30)
unit_price	INTEGER
meat_kind	VARCHAR(20)
has_crust	CHAR(1)
cookedness	INTEGER

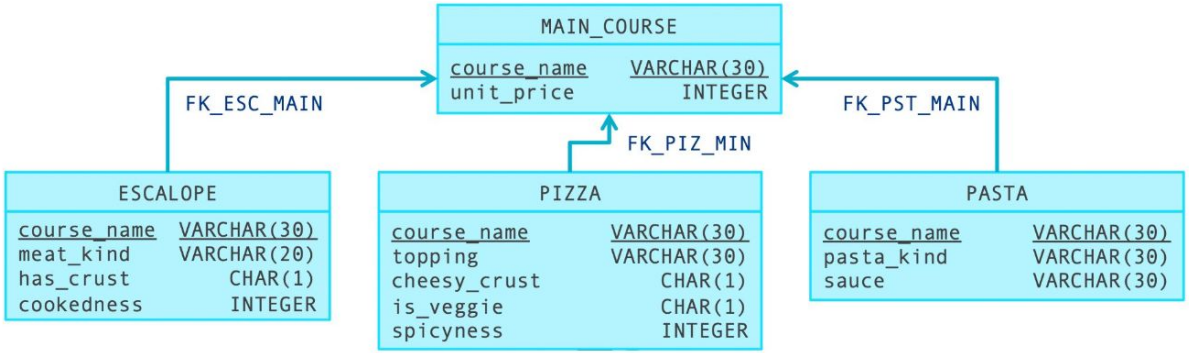
PIZZA	
course_name	VARCHAR(30)
unit_price	INTEGER
topping	VARCHAR(30)
cheesy_crust	CHAR(1)
is_veggie	CHAR(1)
spicyness	INTEGER

PASTA	
course_name	VARCHAR(30)
unit_price	INTEGER
pasta_kind	VARCHAR(30)
sauce	VARCHAR(30)

Development

Generalization

- **Generalization pattern #3**: one relation per subclass + 1 for the parent class (be it abstract or not), with the parent's attributes factored in the parent relation, and foreign keys between the children and the parent
  - > No collateral damage when any of the classes (parent or child) changes
  - > No waste of space even if the parent class has many attributes
  - > No implicit (and error-prone) rule involving NULL values to discriminate records between subclasses
  - > Searching for an object without knowing its concrete type in advance implies looking up the parent and all the children, with a bunch of (possibly slow) outer joins
  - > The parent-child class relationships is explicit in the PDM, thanks to the foreign keys
  - > Unicity of the parent identifier is guaranteed because it becomes the primary key of the parent relation



Data warehousing & BI

Star, snowflake and constellation schema

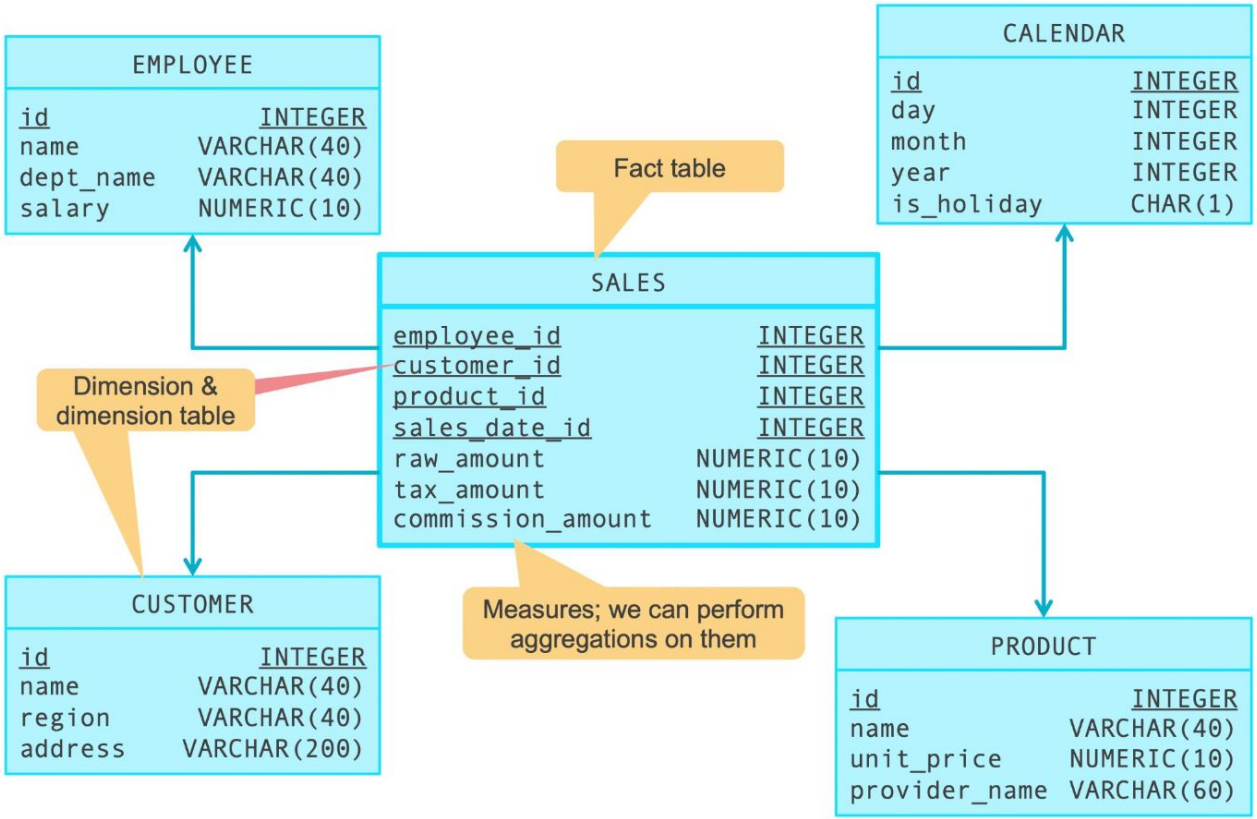
- Problem
  - Data models designed for OLTP workloads are difficult to query in OLAP scenarios, the queries are slow because they involve a lot of complicated joins. Every time a new report must be produced, a specific query has to be designed

- Solution
  - Provide an alternative database schema for reporting. A common choice is the star / snowflake schema, where a central subject table (the **fact table**), holding numeric measures, is surrounded by smaller satellites tables (the **dimension tables**) that are the analysis criteria, giving way to **multidimensional analysis**
  - There can be as many stars or snowflakes as there are subjects of study in the data warehouse/data mart
  - The number of joins is limited by the depth of the tree. Dimensions that are not needed for a specific report are not queried and thus don't need a costly join
  - New reports can be created at will by enumerating the dimension tables. The schema is easy to understand and allows **discovery** of the business domain

- Variations
  - A **star** has one fact table and N dimension tables, the depth of the tree is 1
  - A **snowflake** has one fact table and N dimension tables of arbitrary depth (hierarchies)
  - A **constellation** is a collection of stars / snowflakes where the dimensions are shared between fact tables

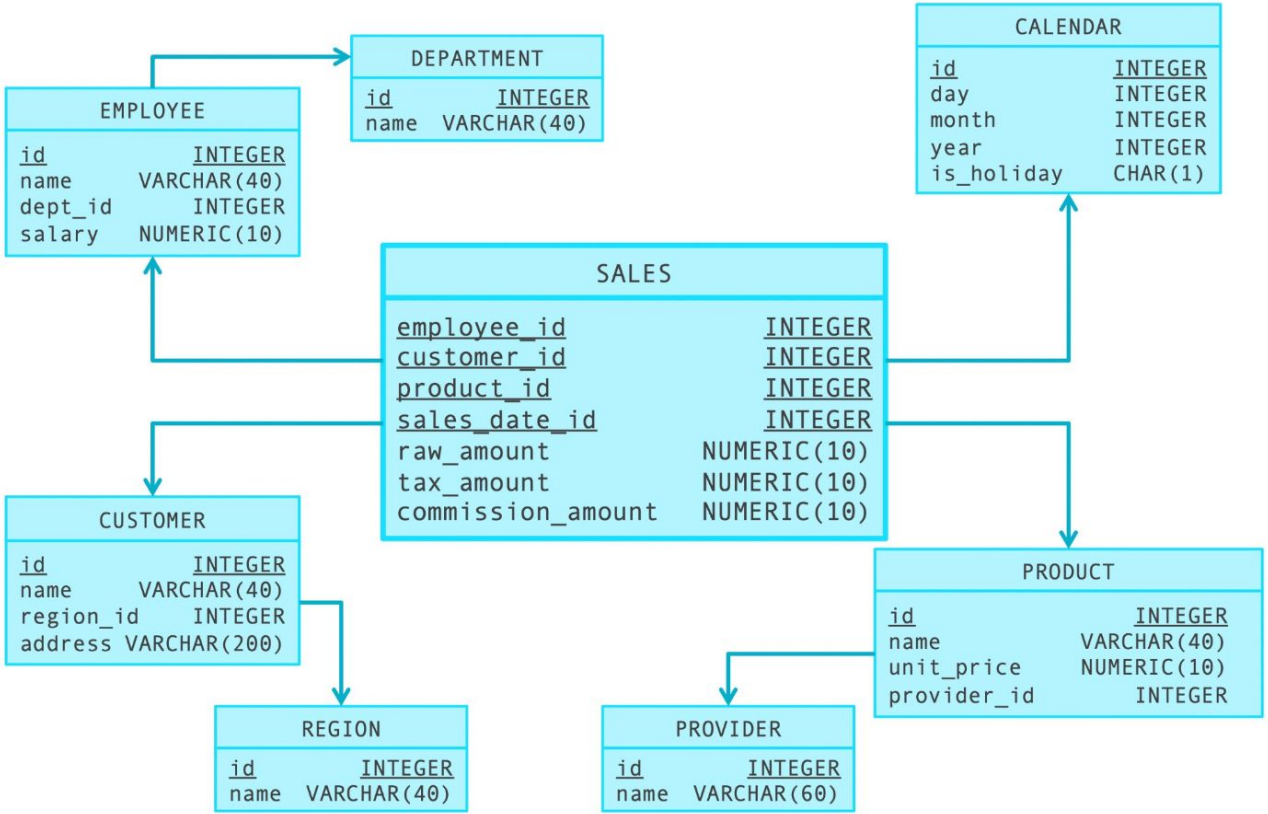
Data warehousing & BI

Example of a star schema



Data warehousing & BI

Example of a snowflake schema



Data warehousing & BI

Denormalization

- Problem
  - > 3NF is necessary to guarantee data consistency, but it leads to lots of tables linked together by foreign keys. As a consequence, is some situations queries are complicated and slow because they involve a lot of joins

- Solution
  - > Abandon 3NF and 2NF by merging tables together (0NF does not help performance and is more related to the key/value store pattern above)

- Variations
  - > This is an optimization strategy and, as such, it is a trade-off between performance and consistency. It should be followed in specific cases where other optimization methods (indices, partitioning, ...) have failed
  - > Denormalization should never be the default choice in a relational context + In NoSQL we are more or less forced to denormalize

customer_id	customer_name	addr_no	customer_address	city
124	John Doe	1	34 Horseradish Ave	NY
125	Mary Smith	1	12 Maniac Bvd	Seattle
125	Mary Smith	2	1 Torvalds Avenue	Boston
125	Mary Smith	3	384 Bull Bvd	NY

DOJO

Sujet

1. Design SQL requests for each question in pgAdmin
2. Implement these requests in the python backend

THE END

See you next week ;-)