Database Design

Fundamentals of a good database design

Black Raven (James Ng)
17 Apr 2021 · 36 min read

4 hours | 13 Videos | 52 Exercises | 19,227 Participants | 4,150 XP

This is a memo to share what I have learnt in Database Design, capturing the learning objectives as well as my personal notes. The course is taught by Lis Sulmont from DataCamp, and it includes 4 chapters:

Chapter 1. Processing, Storing, and Organizing Data
Chapter 2. Database Schemas and Normalization
Chapter 3. Database Views
Chapter 4. Selecting

A good database design is crucial for a high-performance application. Just like you wouldn't start building a house without the benefit of a blueprint, you need to think about how your data will be stored beforehand.

Taking the time to design a database saves time and frustration later on, and a well-designed database ensures ease of access and retrieval of information. While choosing a design, a lot of considerations have to be accounted for.

In this course, you'll learn how to process, store, and organize data in an efficient way. You'll see how to structure data through normalization and present your data with views. Finally, you'll learn how to manage your database and all of this will be done on a variety of datasets from book sales, car rentals, to music reviews.

Chapter 1. Processing, Storing, and Organizing Data

Start your journey into database design by learning about the two approaches to data processing, OLTP and OLAP. In this first chapter, you'll also get familiar with the different forms data can be stored in and learn the basics of data modeling.

OLTP and OLAP

How should we organize and manage data?

- **Schemas:** *How should my data be logically organized?*
- **Normalization:** *Should my data have minimal dependency and redundancy?*
- **Views:** *What joins will be done most often?*
- **Access control:** *Should all users of the data have the same level of access*
- **DBMS:** *How do I pick between all the SQL and noSQL options?*
- and more!

OLTP

Online **T**ransaction **P**rocessing

OLAP

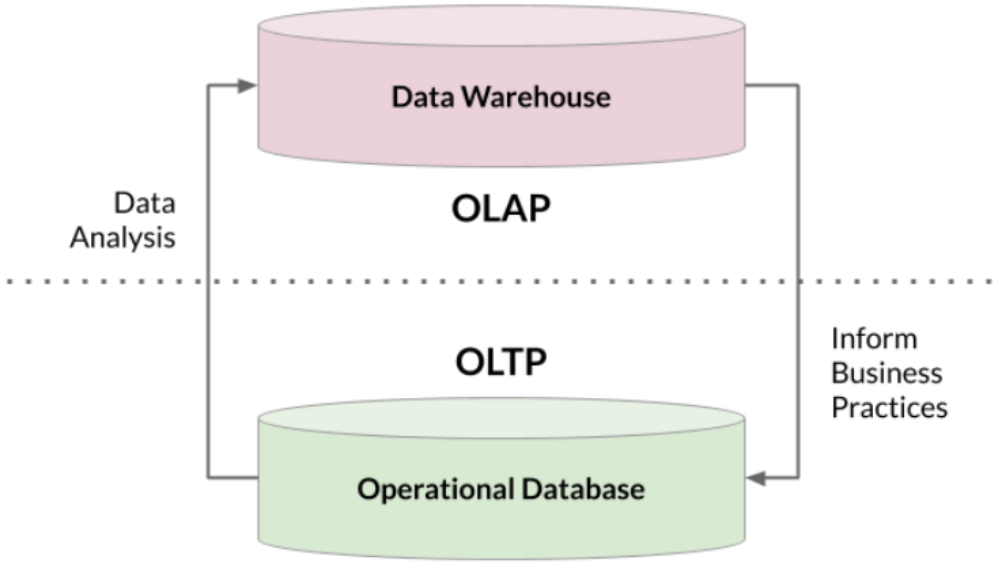
Online **A**nalytical **P**rocessing

- Find the price of a book
- Update latest customer transaction
- Keep track of employee hours

- Calculate books with best profit margin
- Find most loyal customers
- Decide employee of the month

	OLTP	OLAP
Purpose	support daily transactions	report and analyze data
Design	application-oriented	subject-oriented
Data	up-to-date, operational	consolidated, historical
Size	snapshot, gigabytes	archive, terabytes
Queries	simple transactions & frequent updates	complex, aggregate queries & limited updates
Users	thousands	hundreds

OLTP systems are used by more people throughout a company and even a company's customers, while OLAP systems are typically used by only analysts and data scientists at a company.



OLAP and OLTP systems work together; in fact, they need each other. OLTP data is usually stored in an operational database that is pulled and cleaned to create an OLAP data warehouse.

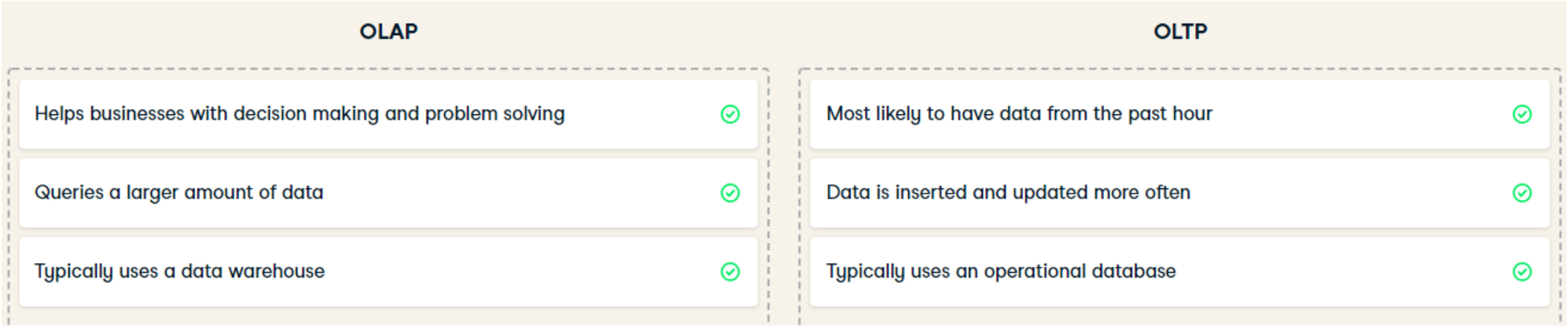
Without transactional data, no analyses can be done in the first place. Analyses from OLAP systems are used to inform business practices and day-to-day activity, thereby influencing the OLTP databases.

Before implementing anything, figure out your business requirements because there are many design decisions you'll have to make. The way you set up your database now will affect how it can be effectively used in the future. Start by figuring out if you need an OLAP or OLTP approach, or perhaps both!

These are the two most common approaches. However, they are not exhaustive, but they are an excellent start to get you on the right path to designing your database.

OLAP vs. OLTP

You should now be familiar with the differences between OLTP and OLAP. In this exercise, you are given a list of cards describing a specific approach which you will categorize between OLAP and OLTP.



Which is better?

The city of Chicago receives many 311 service requests throughout the day. 311 service requests are non-urgent community requests, ranging from graffiti removal to street light outages. Chicago maintains a data repository of all these services organized by type of requests. In this exercise, `Potholes` has been loaded as an example of a table in this repository. It contains pothole reports made by Chicago residents from the past week.

Explore the dataset. What data processing approach is this larger repository most likely using?

Possible Answers

- ○ OLTP because this table could not be used for any analysis.
- ○ OLAP because each record has a unique service request number.
- ◉ OLTP because this table's structure appears to require frequent updates.
- ○ OLAP because this table focuses on pothole requests only.

This table probably uses an OLTP approach because it is updated and holds data from the past week.

Storing data

1. Structured data

- Follows a schema
- Defined data types & relationships

_e.g., SQL, tables in a relational database _

2. Unstructured data

- Schemaless
- Makes up most of data in the world

e.g., photos, chat logs, MP3

3. Semi-structured data

- Does not follow larger schema
- Self-describing structure

e.g., NoSQL, XML, JSON

```
# Example of a JSON file
"user": {
     "profile_use_background_image": true,
     "statuses_count": 31,
     "profile_background_color": "C0DEED",
     "followers_count": 3066,
```

- **Traditional databases**
  - For storing real-time relational structured data = OLTP (transactional)
- **Data warehouses**
  - For analyzing archived relational structured data = OLAP (analytics)
- **Data lakes**
  - For storing data of all structures = flexibility and scalability
  - For analyzing **big data**

Traditional **databases** generally follow relational schemas. Operational databases, which are used for OLTP, are an example of traditional databases. Decades ago, traditional databases used to be enough for data storage. Then as data analytics took off, data **warehouses** were popularized for OLAP approaches. And, now in the age of big data, we need to analyze and store even more data, which is where the **data lake** comes in. The term "traditional databases" is used because many people consider data warehouses and lakes to be a type of database.

Data warehouses are optimized for read-only analytics. They combine data from multiple sources and use massively parallel processing for faster queries. In their database design, they typically use dimensional modeling and a denormalized schema. Amazon, Google, and Microsoft all offer data warehouse solutions, known as Redshift, Big Query, and Azure SQL Data Warehouse, respectively. A **data mart** is a subset of a data warehouse dedicated to a specific topic. Data marts allow departments to have easier access to the data that matters to them.

Warehouses and traditional databases are classified as schema-on-write because the schema is predefined.

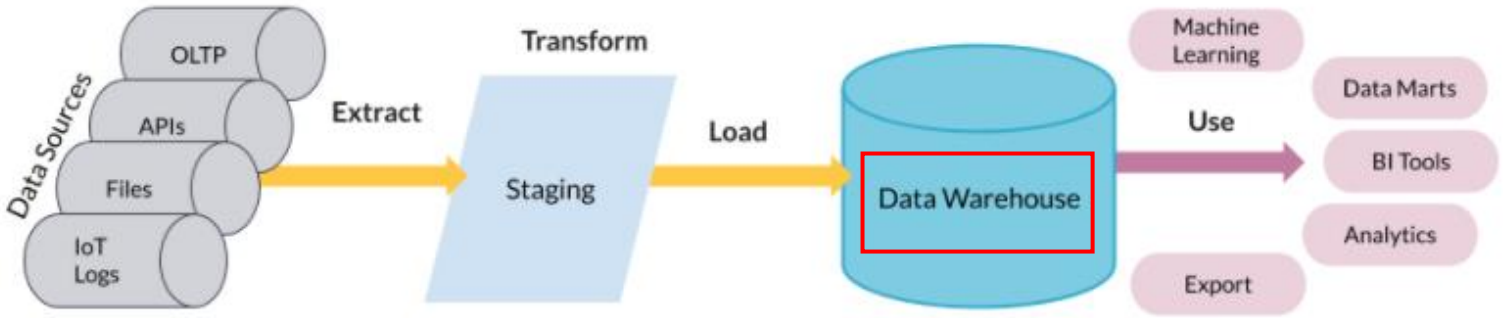
Traditional databases and warehouses can store unstructured data, but not cost-effectively. Data Lake storage is cheaper because it uses object storage as opposed to the traditional block or file storage. This allows massive amounts of data to be stored effectively of all types, from streaming data to operational databases. Lakes are massive because they store all the data that might be used. Data lakes are often petabytes in size - that's 1,000 terabytes! Unstructured data is the most scalable, which permits this size. Lakes are schema-on-read, meaning the schema is created as data is read.

Data lakes have to be organized and cataloged well; otherwise, it becomes an aptly named "data swamp." Data lakes aren't only limited to storage. It's becoming popular to run analytics on data lakes. This is especially true for tasks like deep learning and data discovery, which needs a lot of data that doesn't need to be that "clean."

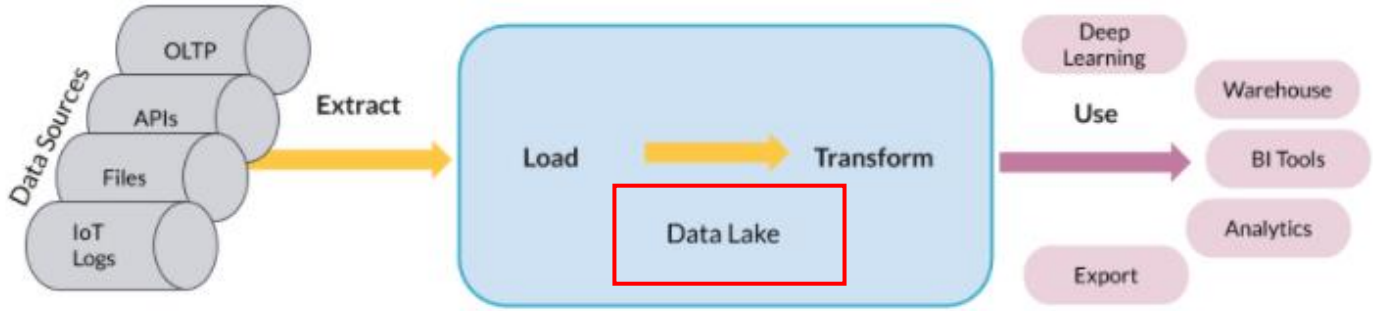
Difference between ETL and ELT

When we think about where to store data, we have to think about how data will get there and in what form. Extract-Transform-Load (ETL) and Extract-Load-Transform (ELT) are two different approaches for describing data flows, with distinct intricacies of building data pipelines.

ETL



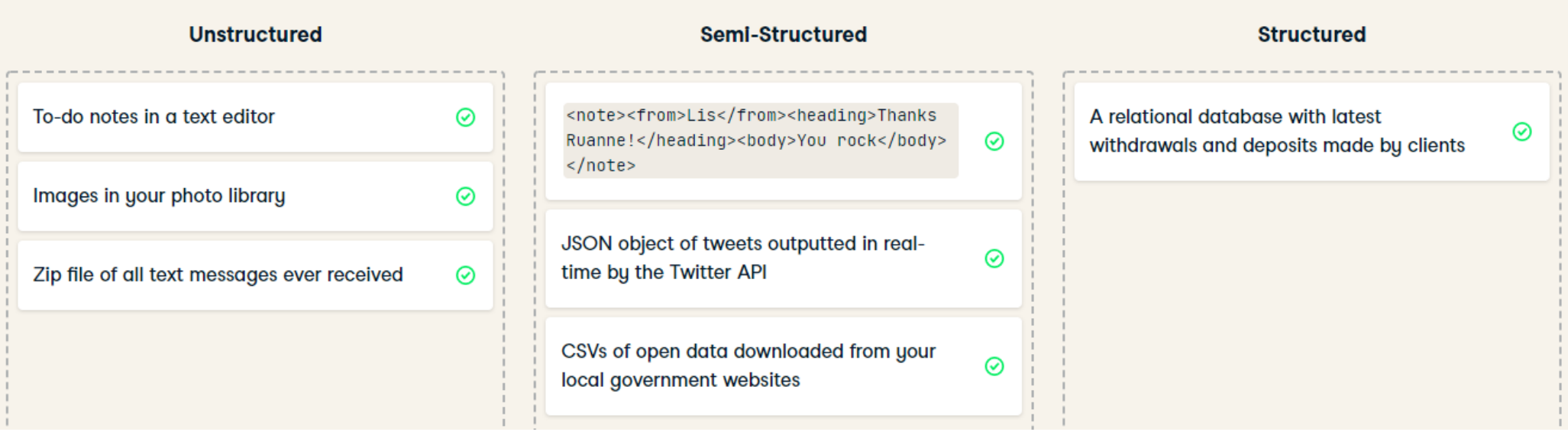
ELT



ETL is the more traditional approach for warehousing and smaller-scale analytics. In ETL, data is transformed before loading into storage - usually to follow the storage's schema, as is the case with warehouses.
But, **ELT** has become common with big data projects. In ELT, the data is stored in its native form in a storage solution like a data lake. Portions of data are transformed for different purposes, from building a data warehouse to doing deep learning.

Name that data type!

In the previous section, you learned about structured, semi-structured, and unstructured data. Structured data is the easiest to analyze because it is organized and cleaned. On the other hand, unstructured data is schemaless, but scales well. In the middle we have semi-structured data for everything in between.

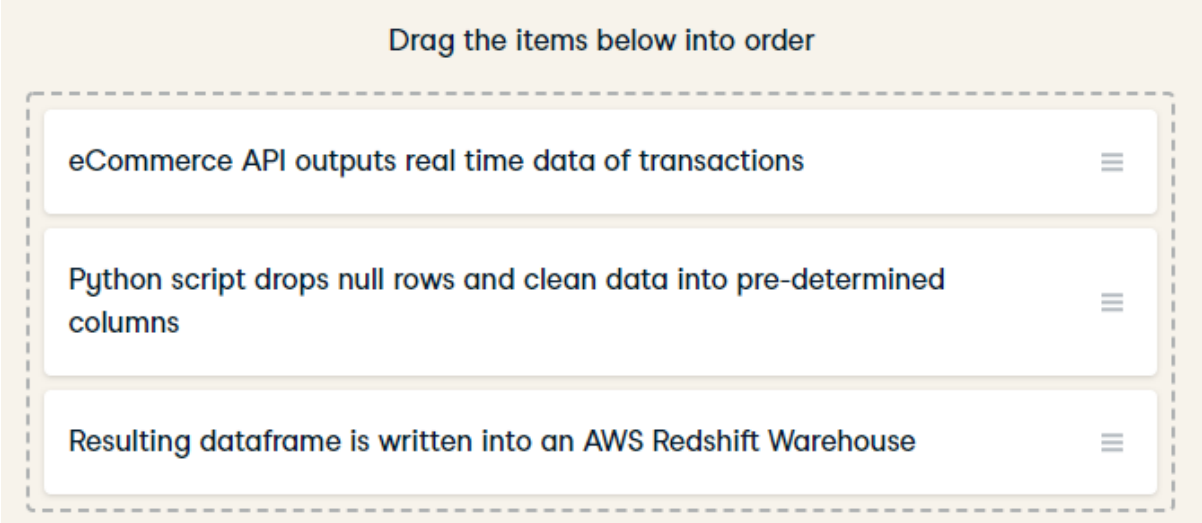


From these real-life examples, can you see why unstructured data is easier to scale than structured data?

Ordering ETL Tasks

You have been hired to manage data at a small online clothing store. Their system is quite outdated because their only data repository is a traditional database to record transactions.

You decide to upgrade their system to a data warehouse after hearing that different departments would like to run their own business analytics. You reason that an ELT approach is unnecessary because there is relatively little data (< 50 GB).



In ETL, raw data is cleaned before being stored. This makes it accessible and ready to use.

Recommend a storage solution

When should you choose a data warehouse over a data lake?

- ○ To train a machine learning model with a 150 GB of raw image data.
- ○ To store real-time social media posts that may be used for future analysis
- ○ To store customer data that needs to be updated regularly
- ◉ To create accessible and isolated data repositories for other analysts

Analysts will appreciate working in a data warehouse more because of its organization of structured data that make analysis easier.

Database design

Database design determines how data is logically stored. There are two important concepts to know when it comes to database design:

- **Database Models** – high-level specifications for database structure. The relational model, which is the most popular, is the model used to make relational databases. It defines rows as records and columns as attributes. It calls for rules such as each row having unique keys.
- **Database Schemas** – a database's blueprint, ie, the implementation of the database model. It takes the logical structure more granularly by defining the specific tables, fields, relationships, indexes, and views a database will have. Schemas must be respected when inserting structured data into a relational database.

Process of creating a *data model* for the data to be stored

1. **Conceptual data model**: describes entities, relationships, and attributes

- *Tools:* data structure diagrams, e.g., entity-relational diagrams and UML diagrams

2. **Logical data model**: defines tables, columns, relationships

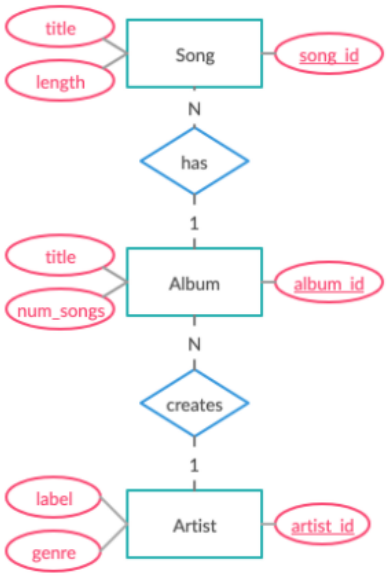
- *Tools:* database models and schemas, e.g., relational model and star schema

3. **Physical data model**: describes physical storage

- *Tools*: partitions, CPUs, indexes, backup systems and tablespaces

These three levels of a data model ensure consistency and provide a plan for implementation and use.

Conceptual - ER diagram

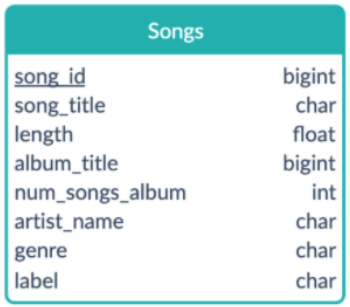


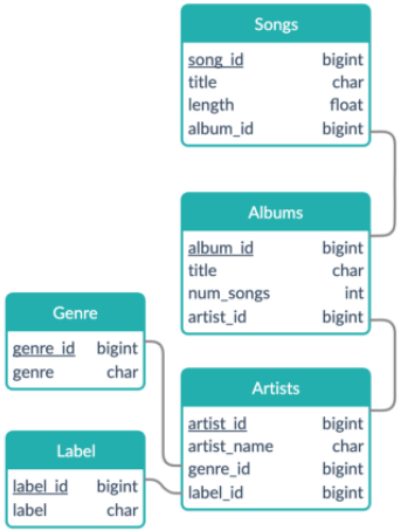
Entities, relationships, and attributes

Logical - schema

Fastest conversion: entities become the tables

Other database design options

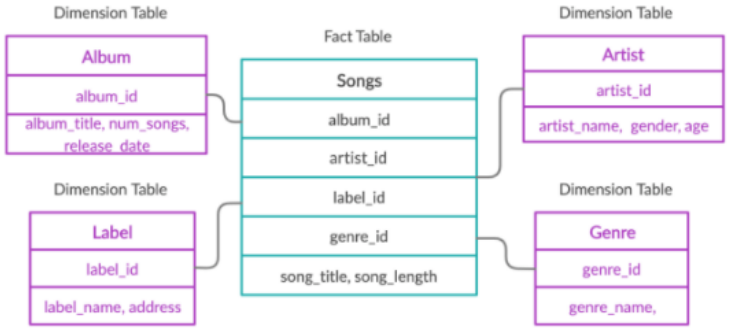




Determining tables

Dimensional modeling is an adaptation of the relational model specifically for data warehouses. It's optimized for **OLAP** type of queries that aim to analyze rather than update. To do this, it uses the star schema. The schema of a dimensional model tends to be easy to interpret and extend.

Elements of dimensional modeling



Organize by:

- What is being analyzed?
- How often do entities change?

Fact tables

- Decided by business use-case
- Holds records of a metric
- Changes regularly
- Connects to dimensions via foreign keys

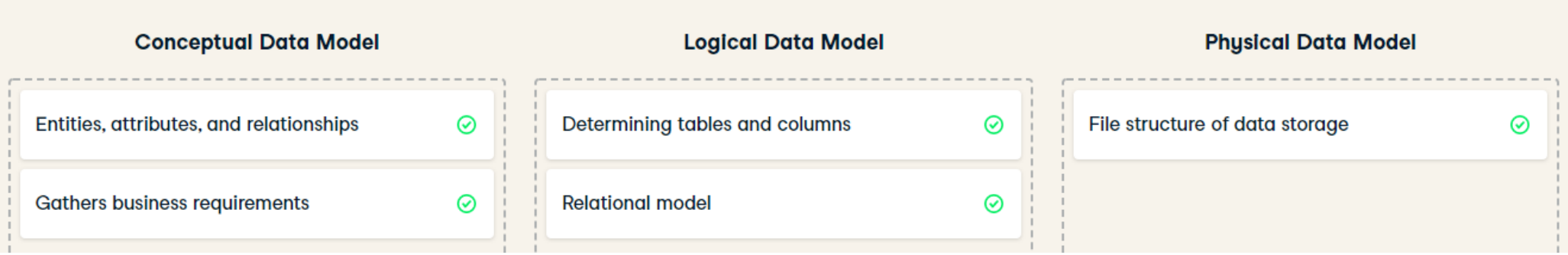
Dimension tables

- Holds descriptions of attributes
- Does not change as often

The records in fact tables often change as new songs get inserted. Albums, labels, artists, and genres will be shared by more than one song - hence records in dimension tables won't change as much.

Classifying data models

In the previous section, we learned about three different levels of data models: conceptual, logical, and physical.



Deciding fact and dimension tables

Imagine that you love running and data. It's only natural that you begin collecting data on your weekly running routine. You're most concerned with tracking how long you are running each week. You also record the route and the distances of your runs. You gather this data and put it into one table called `Runs` with the following schema:

After learning about dimensional modeling, you decide to restructure the schema for the database. `Runs` has been pre-loaded for you.

runs
duration_mins - *float*
week - *int*
month - *varchar(160)*
year - *int*
park_name - *varchar(160)*
city_name - *varchar(160)*
distance_km - *float*
route_name - *varchar(160)*

Out of these possible answers, what would be the best way to organize the fact table and dimensional tables?

- (●) A fact table holding `duration_mins` and foreign keys to dimension tables holding route details and week details, respectively.
- ( ) A fact table holding `week`, `month`, `year` and foreign keys to dimension tables holding route details and duration details, respectively.
- ( ) A fact table holding `route_name`, `park_name`, `distance_km`, `city_name`, and foreign keys to dimension tables holding week details and duration details, respectively.

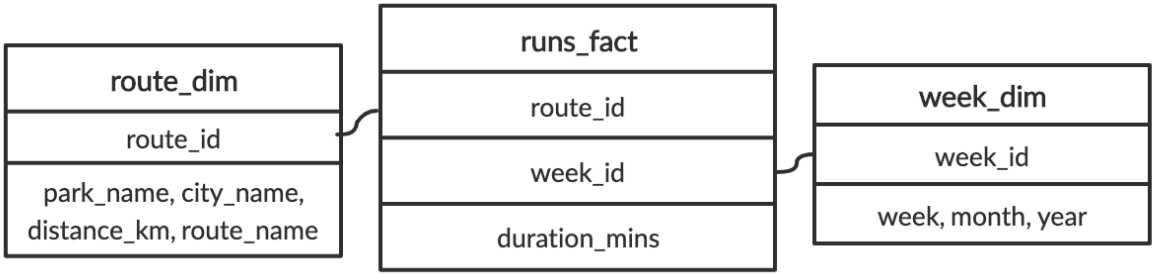
```
-- Create a route dimension table
CREATE TABLE ____(
    route_id INTEGER PRIMARY KEY,
    route_name VARCHAR(160) NOT NULL,
    park_name ____(160) NOT NULL,
    distance_km FLOAT NOT NULL,
    city_name VARCHAR(160) NOT NULL
);
-- Create a week dimension table
CREATE TABLE ____(
    week_id INTEGER PRIMARY KEY,
    week ____ NOT NULL,
    month VARCHAR(160) NOT NULL,
    year ____ NOT NULL
);
```

```
-- Create a route dimension table
CREATE TABLE route(
    route_id INTEGER PRIMARY KEY,
    route_name VARCHAR(160) NOT NULL,
    park_name VARCHAR(160) NOT NULL,
    distance_km FLOAT NOT NULL,
    city_name VARCHAR(160) NOT NULL
);
-- Create a week dimension table
CREATE TABLE week(
    week_id INTEGER PRIMARY KEY,
    week integer NOT NULL,
    month VARCHAR(160) NOT NULL,
    year integer NOT NULL
);
```

The primary keys `route_id` and `week_id` you just created will be foreign keys in the fact table.

Querying the dimensional model

Here it is! The schema reorganized using the dimensional model:



Let's try to run a query based on this schema. How about we try to find the number of minutes we ran in July, 2019? We'll break this up in two steps. First, we'll get the total number of minutes recorded in the database. Second, we'll narrow down that query to `week_id`'s from July, 2019.

```
SELECT
    -- Get the total duration of all runs
    SUM(duration_mins)
FROM
    runs_fact
-- Get all the week_id's that are from July, 2019
INNER JOIN week_dim ON runs_fact.week_id = week_dim.week_id
WHERE month = 'July' and year = '2019';
```

sum
381.46

It looks like you've run 381.46 minutes in July. Because of its structure, the dimensional model usually require queries involving more than one table.

Chapter 2. Database Schemas and Normalization

In this chapter, you will take your data modeling skills to the next level. You'll learn to implement star and snowflake schemas, recognize the importance of normalization and see how to normalize databases to different extents.

Star and snowflake schema

The star schema is the simplest form of the dimensional model. Some use the terms "star schema" and "dimensional model" interchangeably. The star schema is made up of two tables: fact and dimension tables.

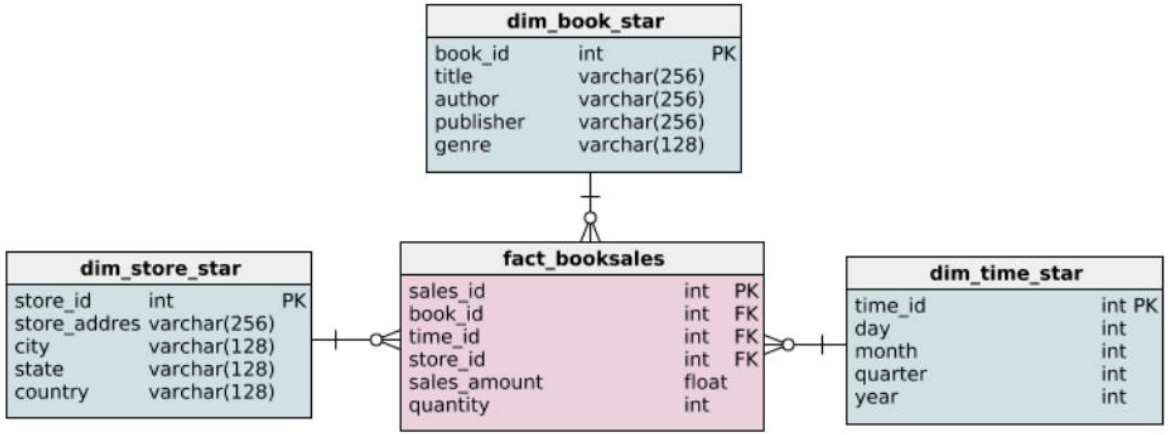
Fact tables

- Holds records of a metric
- Changes regularly
- Connects to dimensions via foreign keys

Dimension tables

- Holds descriptions of attributes
- Does not change as often

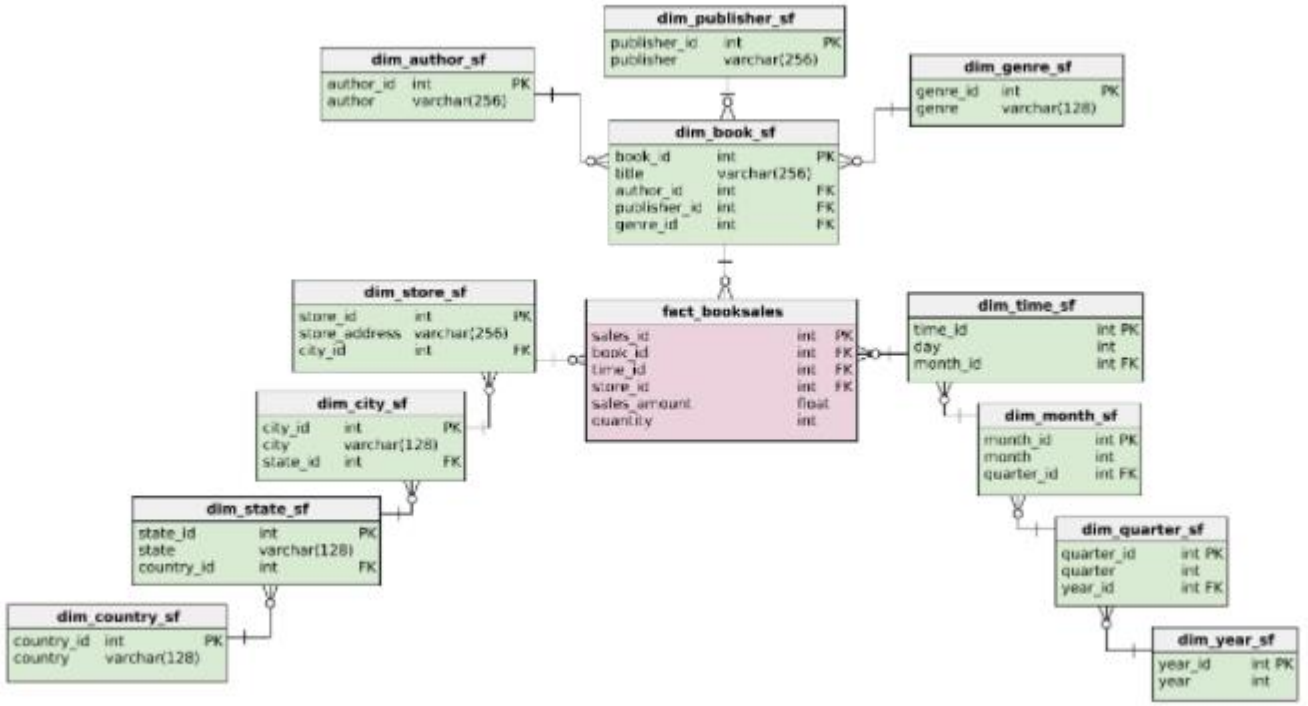
Star schema example

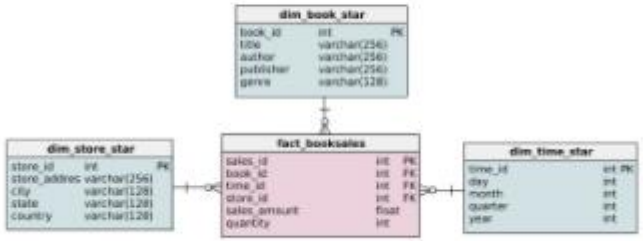
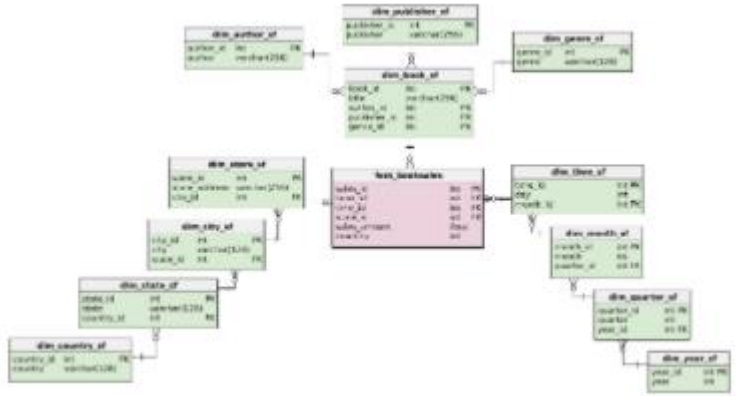


The lines connecting these tables have a special pattern. These lines represent a one-to-many relationship. For example, a store can be part of many book sales, but one sale can only belong to one store.

The snowflake schema is an extension of the star schema. Off the bat, we see that it has more tables. The information contained in this schema is the same as the star schema. In fact, the fact table is the same, but the way the dimension tables are structured is different.

Snowflake schema (an extension)



Star schema: 1 dimension	**Snowflake schemas**: more than 1 dimension
	 Because dimension tables are normalized

Same fact table, different dimensions

What is normalization?

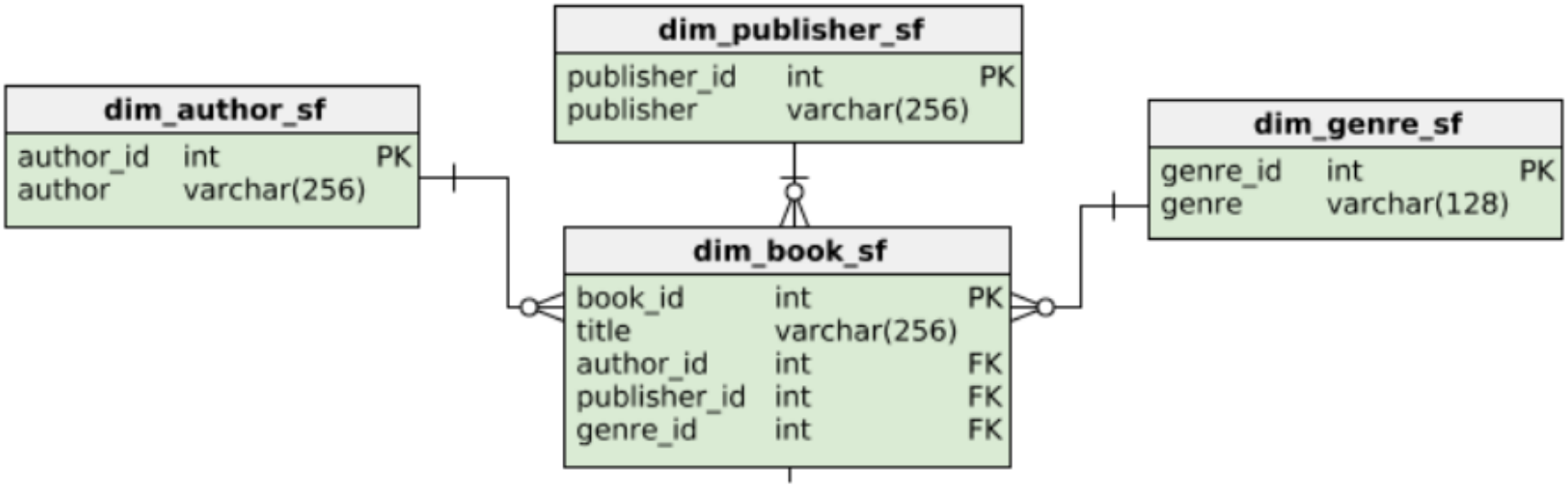
- Database design technique
- Divides tables into smaller tables and connects them via relationships
- **Goal:** reduce redundancy and increase data integrity

The basic idea of normalization is to identify repeating groups of data and create new tables for them.

For a **book**, these fields most likely have repeating values:

- Author
- Publisher
- Genre

We can create new tables for them, and it results in the following snowflake schema:

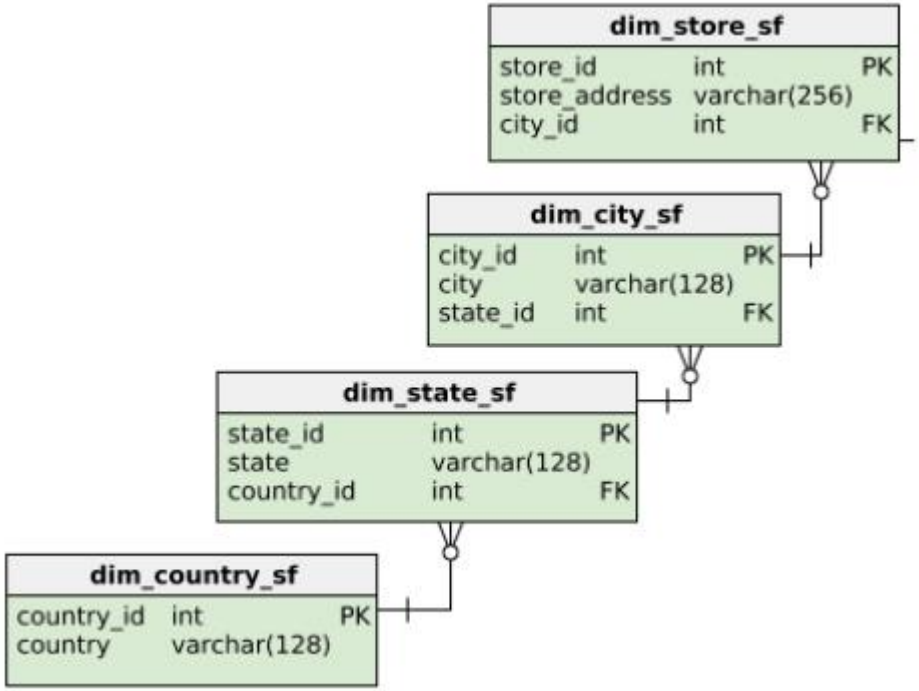


These repeating groups now have their own table

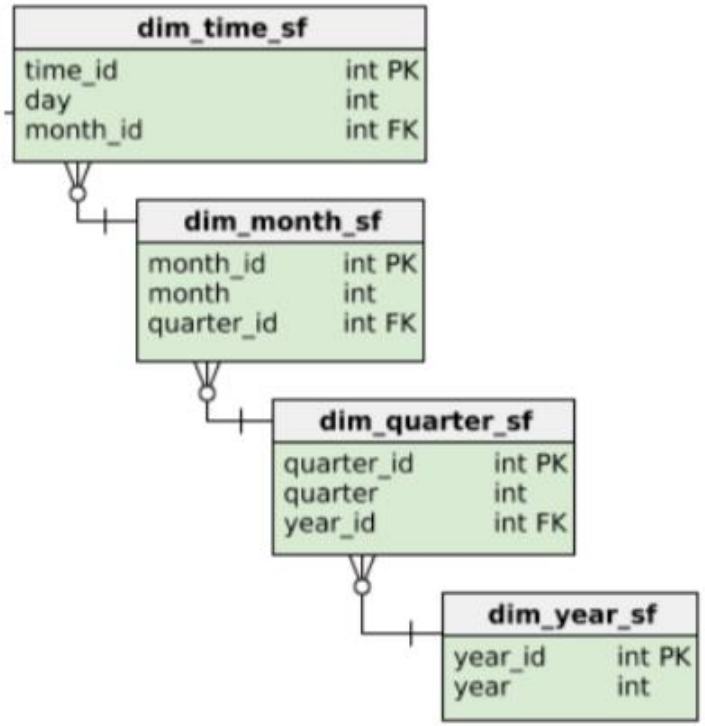
For a **store**, these fields most likely have repeating values:

- City
- State
- Country

Here are the normalized dimension tables representing the book stores:



A city stays in the same state and country; thus, they extend each other over three dimensions.

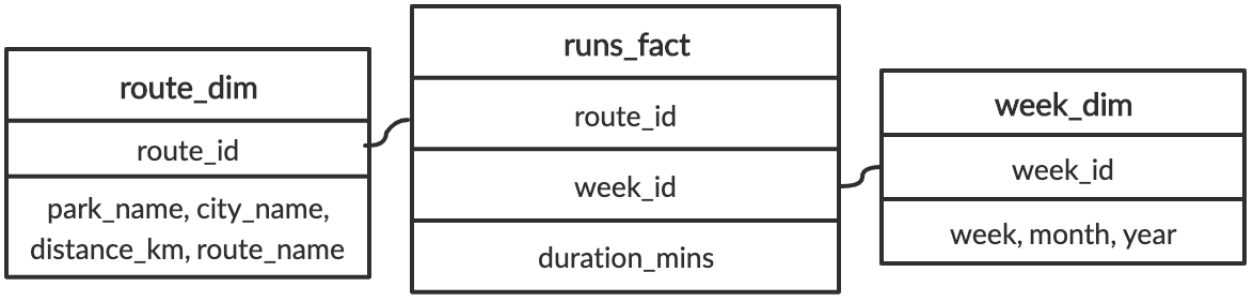


The same is done for the time dimension. A day is part of a month that is part of a quarter.

Put all the normalized dimensions together to get the snowflake schema.

Running from star to snowflake

Remember your running database from last chapter?



After learning about the snowflake schema, you convert the current star schema into a snowflake schema. To do this, you normalize `route_dim` and `week_dim`. The tables `runs_fact`, `route_dim`, and `week_dim` have been loaded.

Which option best describes the resulting new tables after doing this?

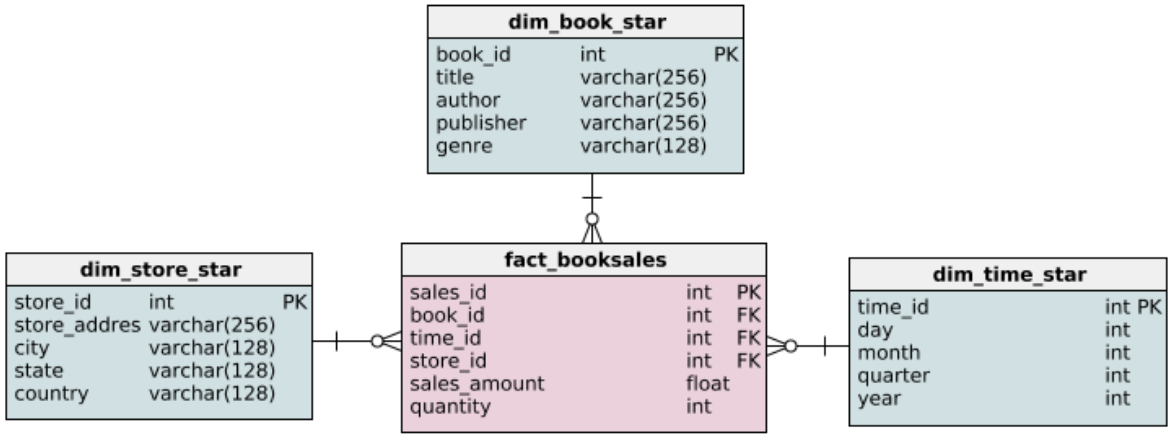
- `week_dim` is extended two dimensions with new tables for `month` and `year`. `route_dim` is extended one dimension with a new table for `city`.
- **`week_dim` is extended two dimensions with new tables for `month` and `year`. `route_dim` is extended two dimensions with new tables for `city` and `park`.**
- `week_dim` is extended three dimensions with new tables for `week`, `month` and `year`. `route_dim` is extended one dimension with new tables for `city` and `park`.

Obviously, month, year, city, and park are indeed repeated often. year and city would extend month and park, respectively.

Adding foreign keys

Foreign key references are essential to both the snowflake and star schema. When creating either of these schemas, correctly setting up the foreign keys is vital because they connect dimensions to the fact table. They also enforce a one-to-many relationship, because unless otherwise specified, a foreign key can appear more than once in a table and primary key can appear only once.

The `fact_booksales` table has three foreign keys: `book_id`, `time_id`, and `store_id`. In this exercise, the four tables that make up the star schema below have been loaded. However, the foreign keys still need to be added.

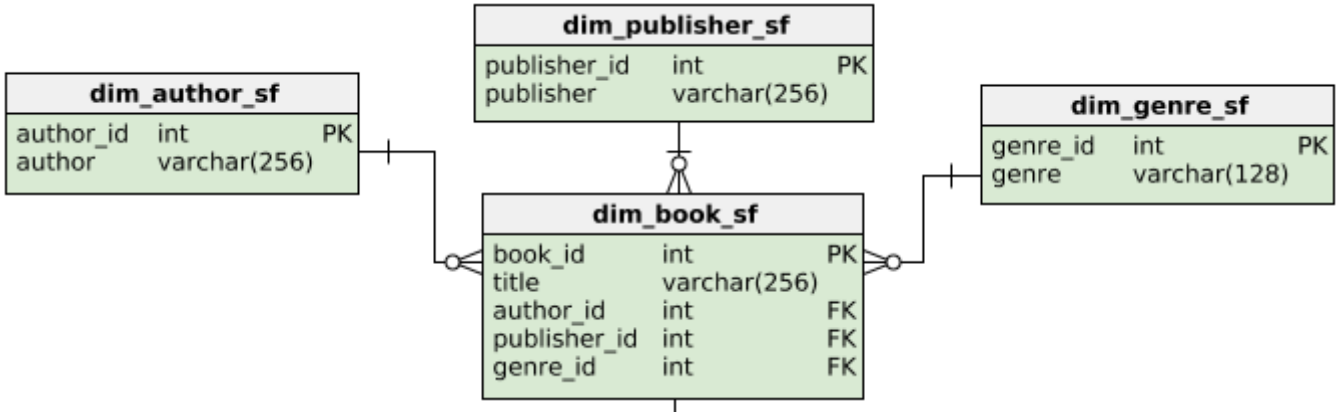



```
-- Add the book_id foreign key
ALTER TABLE fact_booksales ADD CONSTRAINT sales_book
    FOREIGN KEY (book_id) REFERENCES dim_book_star (book_id);

-- Add the time_id foreign key
ALTER TABLE fact_booksales ADD CONSTRAINT sales_time
   FOREIGN KEY (time_id) REFERENCES dim_time_star (time_id);

-- Add the store_id foreign key
ALTER TABLE fact_booksales ADD CONSTRAINT sales_store
   FOREIGN KEY (store_id) REFERENCES dim_store_star (store_id);
```

The foreign keys have been added so now we can ensure data consistency whenever new data is inserted to the database.

Extending the book dimension

In the previous section, we saw how the book dimension differed between the star and snowflake schema. The star schema's dimension table for books, `dim_book_star`, has been loaded and below is the snowflake schema of the book dimension.



In this exercise, you are going to extend the star schema to meet part of the snowflake schema's criteria. Specifically, you will create `dim_author` from the data provided in `dim_book_star`.

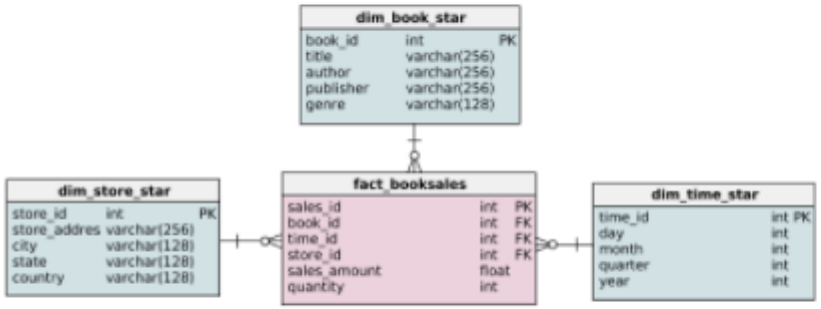
```sql
-- Create a new table for dim_author with an author column
CREATE TABLE dim_author (
    author varchar(256)  NOT NULL
);

-- Insert authors
INSERT INTO dim_author
SELECT DISTINCT author FROM dim_book_star;

-- Add a primary key
ALTER TABLE dim_author ADD COLUMN author_id SERIAL PRIMARY KEY;

-- Output the new table
SELECT * FROM dim_author;
```

author	author_id
Octavia E. Butler	1
F. Scott Fitzgerald	2
Beverly Cleary	3
Barack Obama	4
Alice Waters	5
Agatha Christie	6

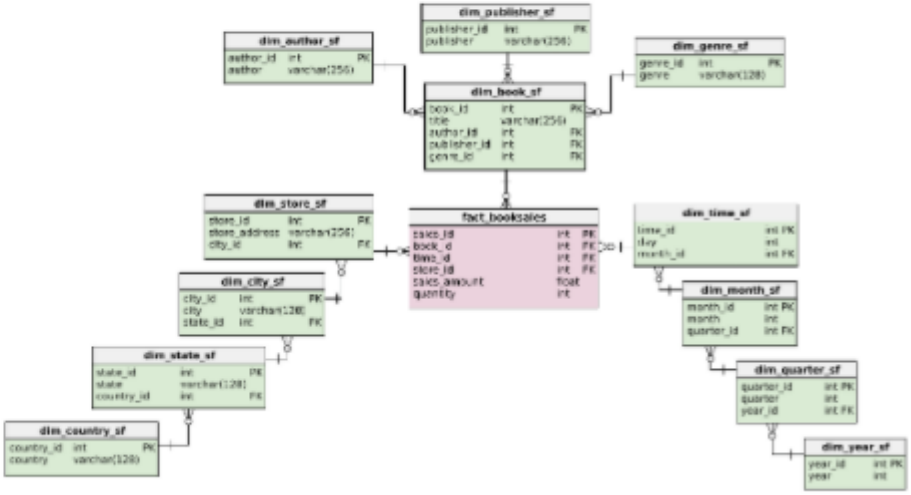
You've created a dimension table that succesfully meets the schema criteria - it has all the authors with no repeats and unique author_ids. If we were to continue completing the star schema, we would need to create tables for the other dimensions using similar code.

Normalized and denormalized databases

Denormalized: star schema



Normalized: snowflake schema



Denormalized Query

Goal: get quantity of all Octavia E. Butler books sold in Vancouver in Q4 of 2018

```
SELECT SUM(quantity) FROM fact_booksales
  -- Join to get city
  INNER JOIN dim_store_star on fact_booksales.store_id = dim_store_star.store_id
  -- Join to get author
  INNER JOIN dim_book_star on fact_booksales.book_id = dim_book_star.book_id
  -- Join to get year and quarter
  INNER JOIN dim_time_star on fact_booksales.time_id = dim_time_star.time_id
WHERE
  dim_store_star.city = 'Vancouver' AND dim_book_star.author = 'Octavia E. Butler' AND
  dim_time_star.year = 2018 AND dim_time_star.quarter = 4;
```

```
7600
```

Total of 3 joins

Normalized query

```
SELECT
  SUM(fact_booksales.quantity)
FROM
  fact_booksales
  -- Join to get city
  INNER JOIN dim_store_sf ON fact_booksales.store_id = dim_store_sf.store_id
  INNER JOIN dim_city ON dim_store_sf.city_id = dim_city_sf.city_id
  -- Join to get author
  INNER JOIN dim_book_sf ON fact_booksales.book_id = dim_book_sf.book_id
  INNER JOIN dim_author_sf ON dim_book_sf.author_id = dim_author_sf.author_id
  -- Join to get year and quarter
  INNER JOIN dim_time_sf ON fact_booksales.time_id = dim_time_sf.time_id
  INNER JOIN dim_month_sf ON dim_time_sf.month_id = dim_month_sf.month_id
  INNER JOIN dim_quarter_sf ON dim_month_sf.quarter_id =  dim_quarter_sf.quarter_id
  INNER JOIN dim_year_sf ON dim_quarter_sf.year_id = dim_year_sf.year_id
```

```
WHERE
  dim_city_sf.city = `Vancouver`
  AND
  dim_author_sf.author = `Octavia E. Butler`
  AND
  dim_year_sf.year = 2018 AND dim_quarter_sf.quarter = 4;
```

```
sum
7600
```

Total of 8 joins

Normalization saves space

dim_store_star

id	store_address	city	state	country
1	67 First St	**Brooklyn**	**New York**	**USA**
2	12 Jefferson Rd	San Francisco	**California**	**USA**
3	90 Coolidge St	Los Angeles	**California**	**USA**
4	85 Main Ave	**Brooklyn**	**New York**	**USA**
5	123 Bedford St	**Brooklyn**	**New York**	**USA**

dim_store_sf

id	store_address	city_id
1	67 First St	2
2	12 Jefferson Rd	3
3	90 Coolidge St	4
4	85 Main Ave	2
5	123 Bedford St	2

dim_city_sf

city_id	city_name	state_id
2	Brooklyn	43
3	San Francisco	36
4	Los Angeles	36

dim_state_sf

state_id	state	country_id
43	New York	121
36	California	121

Normalization eliminates **data redundancy**

Normalization ensures better data integrity

1. Enforces data consistency

Must respect naming conventions because of referential integrity, e.g., 'California', not 'CA' or 'california'

2. Safer updating, removing, and inserting

Less data redundancy = less records to alter

3. Easier to redesign by extending

Smaller tables are easier to extend than larger tables

Database normalization

Advantages

- Normalization eliminates data redundancy: save on storage
- Better data integrity: accurate and consistent data

Disadvantages

- Complex queries require more CPU

OLTP

e.g., Operational databases

Typically highly normalized

- Write-intensive
- Prioritize quicker and safer insertion of data

OLAP

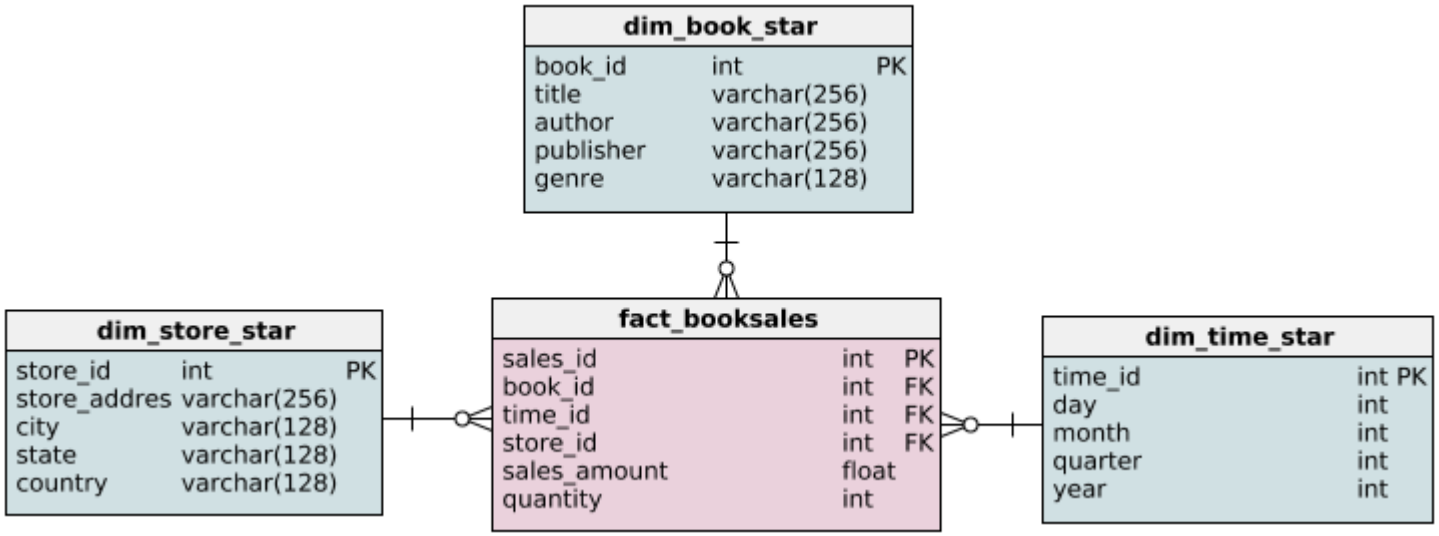
e.g., Data warehouses

Typically less normalized

- Read-intensive
- Prioritize quicker queries for analytics

Querying the star schema

The novel genre hasn't been selling as well as your company predicted. To help remedy this, you've been tasked to run some analytics on the novel genre to find which areas the Sales team should target. To begin, you want to look at the total amount of sales made in each state from books in the novel genre. Luckily, you've just finished setting up a data warehouse with the following star schema:



The tables from this schema have been loaded.

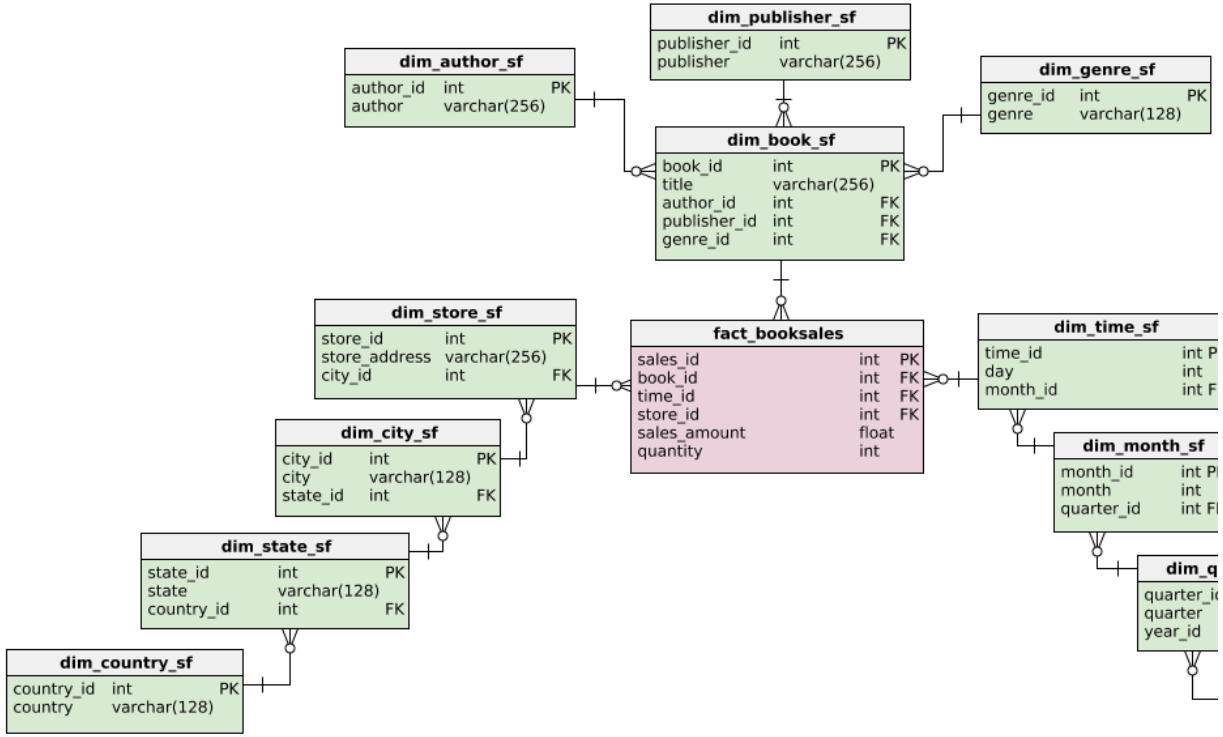
```sql
-- Output each state and their total sales_amount
SELECT dim_store_star.state, sum(sales_amount)
FROM fact_booksales
    -- Join to get book information
    JOIN dim_book_star on fact_booksales.book_id = dim_book_star.book_id
    -- Join to get store information
    JOIN dim_store_star on fact_booksales.store_id = dim_store_star.store_id
-- Get all books with in the novel genre
WHERE
    dim_book_star.genre = 'novel'
-- Group results by state
GROUP BY
    dim_store_star.state;
```

state	sum
Florida	295594.2
Vermont	216282
Louisiana	176979
New York	248529.6
New Jersey	272956.2
British Columbia	374629.2

We now have a nice list of the amount of money made from novels in each state. Note that it took only two joins to run this query.

Querying the snowflake schema

Imagine that you didn't have the data warehouse set up. Instead, you'll have to run this query on the company's operational database, which means you'll have to rewrite the previous query with the following snowflake schema:



The tables in this schema have been loaded. Remember, our goal is to find the amount of money made from the novel genre in each state.

```
-- Output each state and their total sales_amount
SELECT dim_state_sf.state, sum(sales_amount)
FROM fact_booksales
    -- Joins for the genre
```

```
    JOIN dim_book_sf on fact_booksales.book_id = dim_book_sf.book_id
    JOIN dim_genre_sf on dim_book_sf.genre_id = dim_genre_sf.genre_id
    -- Joins for the state
    JOIN dim_store_sf on fact_booksales.store_id = dim_store_sf.store_id
    JOIN dim_city_sf on dim_store_sf.city_id = dim_city_sf.city_id
    JOIN dim_state_sf on  dim_city_sf.state_id = dim_state_sf.state_id
-- Get all books with in the novel genre and group the results by state
WHERE
    dim_genre_sf.genre = 'novel'
GROUP BY
   dim_state_sf.state;
```

state	sum
British Columbia	374629.2
California	583248.6
Florida	295594.2
Louisiana	176979
Massachusetts	344671.8

Showing 14 out of 14 rows

This query was definitely more work than the previous one. It wouldn't be practical to have to think about all these joins if you're doing a lot of analytics.

Updating countries

Going through the company data, you notice there are some inconsistencies in the store addresses. These probably occurred during data entry, where people fill in fields using different naming conventions. This can be especially seen in the `country` field, and you decide that countries should be represented by their abbreviations. The only countries in the database are Canada and the United States, which should be represented as `USA` and `CA`.

In this exercise, you will compare the records that need to be updated in order to do this task on the star and snowflake schema. `dim_store_star` and `dim_country_sf` have been loaded.

Output all the records that need to be updated in the **star schema** so that countries are represented by their abbreviations.

```
-- Output records that need to be updated in the star schema
SELECT * FROM dim_store_star
WHERE country != 'USA' AND country !='CA';
```

store_id	store_address	city	state	country
798	23 Jeanne Ave	Montreal	Quebec	Canada
799	56 University St	Quebec City	Quebec	Canada
800	23 Verte Ave	Montreal	Quebec	Canada
801	33 Smith St	Toronto	Ontario	Canada
802	44 Green Blvd	Toronto	Ontario	Canada
803	55 Hover Lane	Ottawa	Ontario	Canada
804	68 Washington St	Vancouver	British Columbia	Canada

Showing 18 out of 18 rows

How many records would need to be updated in the **snowflake schema**?

Possible Answers

- 18 records
- 2 records
- 1 record
- 0 records

Only one record needs to be changed - Canada to CA. Updating is typically simpler in a snowflake schema because there are less records to update because redundant values are minimized to their own table (e.g., countries have their own table, dim_country_sf). Snowflake schemas are also better at enforcing naming conventions due to referential integrity. Note how there weren't any variations in how Canada and USA were referred to in the snowflake schema.

Extending the snowflake schema

The company is thinking about extending their business beyond bookstores in Canada and the US. Particularly, they want to expand to a new continent. In preparation, you decide a `continent` field is needed when storing the addresses of stores.

Luckily, you have a snowflake schema in this scenario. As we discussed in the previous section, the snowflake schema is typically faster to extend while ensuring data consistency. Along with `dim_country_sf`, a table called `dim_continent_sf` has been loaded. It contains the only continent currently needed, `North America`, and a primary key. In this exercise, you'll need to extend `dim_country_sf` to reference `dim_continent_sf`.

```
-- Add a continent_id column with default value of 1
ALTER TABLE dim_country_sf
ADD continent_id int NOT NULL DEFAULT(1);

-- Add the foreign key constraint
ALTER TABLE dim_country_sf ADD CONSTRAINT country_continent
```

```
   FOREIGN KEY (continent_id) REFERENCES dim_continent_sf(continent_id);

-- Output updated table
SELECT * FROM dim_country_sf;
```

country_id	country	continent_id
1	Canada	1
2	USA	1

We have successfully extended the snowflake schema to have continents. That wasn't too bad as it only required altering one table and we can be sure of data consistency. This type of extension is a big benefit of the snowflake schema.

Normal forms

Normalization

Identify repeating groups of data and create new tables for them

A more formal definition:

> The goals of normalization are to:
>
> - Be able to characterize the level of redundancy in a relational schema
> - Provide mechanisms for transforming schemas in order to remove redundancy

Normal forms (NF)

Ordered from least to most normalized:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Elementary key normal form (EKNF)
- Boyce-Codd normal form (BCNF)
- Fourth normal form (4NF)
- Essential tuple normal form (ETNF)
- Fifth normal form (5NF)
- Domain-key Normal Form (DKNF)
- Sixth normal form (6NF)

1NF rules

- Each record must be unique - no duplicate rows
- Each cell must hold one value

Initial data

Student_id	Student_Email	Courses_Completed
235	jim@gmail.com	Introduction to Python, Intermediate Python
455	kelly@yahoo.com	Cleaning Data in R
767	amy@hotmail.com	Machine Learning Toolbox, Deep Learning in Python

In 1NF form

Student_id	Student_Email
235	jim@gmail.com
455	kelly@yahoo.com
767	amy@hotmail.com

Student_id	Completed
235	Introduction to Python
235	Intermediate Python
455	Cleaning Data in R
767	Machine Learning Toolbox
767	Deep Learning in Python

Goal: all the records are unique, and each column has 1 value

2NF

- Must satisfy 1NF **AND**
  - If primary key is one column
    - then automatically satisfies 2NF
  - If there is a composite primary key
    - then each non-key column must be dependent on all the keys

Initial data

Student_id (PK)	Course_id (PK)	Instructor_id	Instructor	Progress
235	2001	560	Nick Carchedi	.55
455	2345	658	Ginger Grant	.10
767	6584	999	Chester Ismay	1.00

In 2NF form

Student_id (PK)	Course_id (PK)	Percent_Completed
235	2001	.55
455	2345	.10
767	6584	1.00

Course_id (PK)	Instructor_id	Instructor
2001	560	Nick Carchedi
2345	658	Ginger Grant
6584	999	Chester Ismay

3NF

- Satisfies 2NF
- No **transitive dependencies**: non-key columns can't depend on other non-key columns

Initial Data

Course_id (PK)	Instructor_id	Instructor	Tech
2001	560	Nick Carchedi	Python
2345	658	Ginger Grant	SQL
6584	999	Chester Ismay	R

In 3NF

Course_id (PK)	Instructor	Tech
2001	Nick Carchedi	Python
2345	Ginger Grant	SQL
6584	Chester Ismay	R

Instructor_id	Instructor
560	Nick Carchedi
658	Ginger Grant
999	Chester Ismay

Data anomalies

What is risked if we don't normalize enough?

1. Update anomaly

2. Insertion anomaly

3. Deletion anomaly

Update anomaly

Data inconsistency caused by data redundancy when updating

Student_ID	Student_Email	Enrolled_in	Taught_by
230	lisa@gmail.com	Cleaning Data in R	Maggie Matsui
367	bob@hotmail.com	Data Visualization in R	Ronald Pearson
520	ken@yahoo.com	Introduction to Python	Hugo Bowne-Anderson
520	ken@yahoo.com	Arima Models in R	David Stoffer

To update student `520` 's email:

- Need to update more than one record, otherwise, there will be inconsistency
- User updating needs to know about redundancy

Insertion anomaly

Unable to add a record due to missing attributes

Student_ID	Student_Email	Enrolled_in	Taught_by
230	lisa@gmail.com	Cleaning Data in R	Maggie Matsui
367	bob@hotmail.com	Data Visualization in R	Ronald Pearson
520	ken@yahoo.com	Introduction to Python	Hugo Bowne-Anderson
520	ken@yahoo.com	Arima Models in R	David Stoffer

Unable to insert a student who has signed up but not enrolled in any courses

Deletion anomaly

Deletion of record(s) causes unintentional loss of data

Student_ID	Student_Email	Enrolled_in	Taught_by
230	lisa@gmail.com	Cleaning Data in R	Maggie Matsui
367	bob@hotmail.com	Data Visualization in R	Ronald Pearson
520	ken@yahoo.com	Introduction to Python	Hugo Bowne-Anderson
520	ken@yahoo.com	Arima Models in R	David Stoffer

If we delete Student `230`, what happens to the data on `Cleaning Data in R`?

Data anomalies

What is risked if we don't normalize enough?

1. Update anomaly

2. Insertion anomaly

3. Deletion anomaly

The more normalized the database, the less prone it will be to data anomalies

Converting to 1NF

In the next three exercises, you'll be working through different tables belonging to a car rental company. Your job is to explore different schemas and gradually increase the normalization of these schemas through the different normal forms. At this stage, we're not worried about relocating the data, but rearranging the tables.

A table called `customers` has been loaded, which holds information about customers and the cars they have rented.

```
select * from customers
```

customer_id	customer_name	cars_rented	invoice_id	premium_member	salutation
1453	Kelly Brennan	4KL298	4534	false	Dr
1454	Tom Nguyen	5PL4YY	9832	false	Mr
1455	Georgia Kim	5H9OP5, 9PH8GF, 499ERW	2903, 3490, 1021	true	Ms
1456	Jean Ford	4KL298, 9PH8GF	7890, 4494	true	Mrs

Showing 4 out of 4 rows

Does the `customers` table meet 1NF criteria?

- ○ Yes, all the records are unique.
- ◉ No, because there are multiple values in `cars_rented` and `invoice_id`
- ○ No, because the non-key columns such as don't depend on `customer_id`, the primary key.

Drop two columns from customers table to satisfy 1NF

```sql
-- Create a new table to hold the cars rented by customers
CREATE TABLE cust_rentals (
  customer_id INT NOT NULL,
  car_id VARCHAR(128) NULL,
  invoice_id VARCHAR(128) NULL
);

-- Drop column from customers table to satisfy 1NF
ALTER TABLE customers
DROP COLUMN cars_rented,
DROP COLUMN invoice_id;
```

We now have two tables: (1) customers which holds customer information and (2) cust_rentals which holds the car_ids rented by different customer_ids. This satisfies 1NF. In a real situation, we would need to fill the new table before dropping any columns.

Converting to 2NF

Let's try normalizing a bit more. In the last exercise, you created a table holding `customer_id`s and `car_id`s. This has been expanded upon and the resulting table, `customer_rentals`, has been loaded for you. Since you've got 1NF down, it's time for 2NF.

```
select * from customer_rentals
```

customer_id	car_id	start_date	end_date	model	manufacturer	type_car	condition	color
1453	4KL298	2019-01-08	2019-01-10	Golf 2017	Volkswagen	hatchback	fair	blue
1454	5PL4YY	2019-03-18	2019-03-21	Camaro 2019	Chevrolet	convertible	excellent	red
1455	5H9OP5	2019-04-14	2019-04-14	CRV 2018	Honda	SUV	good	grey
1455	5H9OP5	2019-05-02	2019-05-16	CRV 2018	Honda	SUV	good	grey
1455	499ERW	2019-01-12	2019-01-13	CRV 2018	Honda	SUV	excellent	black
1456	4KL298	2019-02-17	2019-02-22	Golf 2017	Volkswagen	hatchback	fair	blue
1456	4KL298	2019-03-05	2019-03-20	Golf 2017	Volkswagen	hatchback	fair	blue

Showing 7 out of 7 rows

Why doesn't `customer_rentals` meet 2NF criteria?

- ○ Because the `end_date` doesn't depend on all the primary keys.
- ○ Because there can only be at most two primary keys.
- ◉ Because there are non-key attributes describing the car that only depend on one primary key, `car_id`.

Create a new table for the non-key columns that were conflicting with 2NF criteria. Drop those non-key columns from `customer_rentals`.

```
-- Create a new table to satisfy 2NF
CREATE TABLE cars (
  car_id VARCHAR(256) NULL,
  model VARCHAR(128),
  manufacturer VARCHAR(128),
  type_car VARCHAR(128),
  condition VARCHAR(128),
  color VARCHAR(128)
);

-- Drop columns in customer_rentals to satisfy 2NF
ALTER TABLE customer_rentals
DROP COLUMN model,
DROP COLUMN manufacturer,
DROP COLUMN type_car,
DROP COLUMN condition,
DROP COLUMN color;
```

There we go! model, manufacturer, type_car, conditions, and colors depend on car_id, but are independent of the other two primary keys, customer_id and start_date. The customer or start date cannot change these attributes. Hence, we have put these columns in a new table and dropped them from customer_rentals.

Converting to 3NF

Last, but not least, we are at 3NF. In the last exercise, you created a table holding `car_ids`s and car attributes. This has been expanded upon. For example, `car_id` is now a primary key. The resulting table, `rental_cars`, has been loaded for you.

```
select * from rental_cars
```

car_id	model	manufacturer	type_car	condition	color
4KL298	Golf 2017	Volkswagen	hatchback	fair	blue
5PL4YY	Camaro 2019	Chevrolet	convertible	excellent	red
5H9OP5	CRV 2018	Honda	SUV	good	grey
499ERW	CRV 2018	Honda	SUV	excellent	black

Showing 4 out of 4 rows

Why doesn't `rental_cars` meet 3NF criteria?

- (●) Because there are two columns that depend on the non-key column, `model`.
- ( ) Because there are two columns that depend on the non-key column, `color`.
- ( ) Because 2NF criteria isn't satisfied.

Create a new table for the non-key columns that were conflicting with 3NF criteria. Drop those non-key columns from `rental_cars`.

```
-- Create a new table to satisfy 3NF
CREATE TABLE car_model(
  model VARCHAR(128),
  manufacturer VARCHAR(128),
  type_car VARCHAR(128)
);

-- Drop columns in rental_cars to satisfy 3NF
ALTER TABLE rental_cars
DROP COLUMN condition,
DROP COLUMN color;
```

Can you see how these 3NF tables help reduce data redundancy and potential data anomalies?

Chapter 3. Database Views

Get ready to work with views! In this chapter, you will learn how to create and query views. On top of that, you'll master more advanced capabilities to manage them and end by identifying the difference between materialized and non-materialized views.

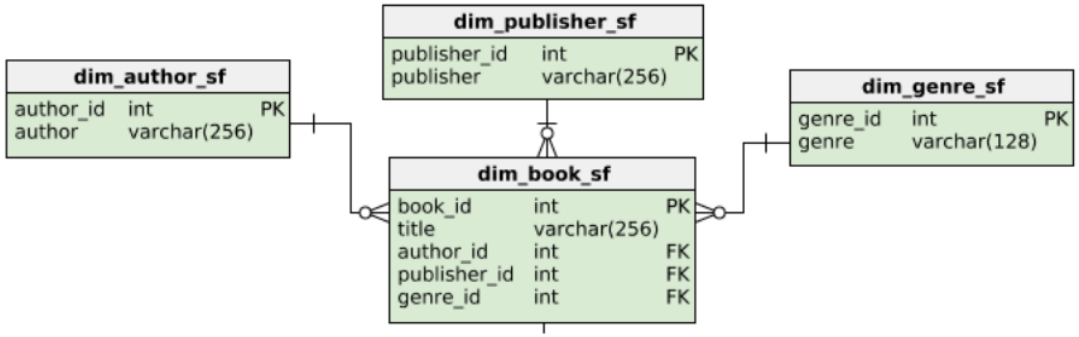
Database views

> In a database, a **view** is the result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object *(Wikipedia)*

Views are virtual tables that are not part of the physical schema.

- A view isn't stored in physical memory; instead, the query to create the view is.
- The data in a view comes from data in tables of the same database.
- Once a view is created, you can query it like a regular table.
- The benefit of a view is that you don't need to retype common queries. It allows you to add virtual tables without altering the database's schema.

Views are simple to create. You take the query of interest and add a line (CREATE VIEW statement) before it to name the view:




```sql
CREATE VIEW scifi_books AS

SELECT title,  author, genre
FROM dim_book_sf
JOIN dim_genre_sf ON dim_genre_sf.genre_id = dim_book_sf.genre_id
JOIN dim_author_sf ON dim_author_sf.author_id = dim_book_sf.author_id
WHERE dim_genre_sf.genre = 'science fiction';
```

After executing the code from the last slide, you can query the view.

```
SELECT * FROM scifi_books
```

```
| title                          | author           | genre           |
|--------------------------------|------------------|-----------------|
| The Naked Sun                  | Isaac Asimov     | science fiction |
| The Robots of Dawn             | Isaac Asimov     | science fiction |
| The Time Machine               | H.G. Wells       | science fiction |
| The Invisible Man              | H.G. Wells       | science fiction |
| The War of the Worlds          | H.G. Wells       | science fiction |
| Wild Seed (Patternmaster, #1) | Octavia E. Butler | science fiction |
| ...                            | ...              | ...             |
```

Behind the scenes

```
SELECT * FROM scifi_books
```

=

```
SELECT * FROM
(SELECT title,  author, genre
FROM dim_book_sf
JOIN dim_genre_sf ON dim_genre_sf.genre_id = dim_book_sf.genre_id
JOIN dim_author_sf ON dim_author_sf.author_id = dim_book_sf.author_id
WHERE dim_genre_sf.genre = 'science fiction');
```

scifi_books isn't a real table with physical memory. When we run this select statement, the query above is actually being run.

To get all the views in your database, you can run a query on:

```
SELECT * FROM INFORMATION_SCHEMA.views;
```

Includes system views

```
SELECT * FROM information_schema.views
WHERE table_schema NOT IN ('pg_catalog', 'information_schema');
```

Excludes system views

Note that this command is specific to PostgreSQL.

Benefits of views

- Doesn't take up storage
- A form of **access control**
  - Hide sensitive columns and restrict what user can see
- Masks complexity of queries
  - Useful for highly normalized schemas

Use this DB structure for exercises:

Tables vs. views

Views have been described as "virtual tables". It's true that views are similar to tables in certain aspects, but there are key differences. In this exercise, you will organize these differences and similarities.

Only Tables	Views & Tables	Only Views
Part of the physical schema of a database.	Contains rows and columns	Takes up less memory
	Can be queried	Always defined by a query
	Has access control	

These characteristics can help make decisions about what needs to be a table or view.

Viewing views

Because views are very useful, it's common to end up with many of them in your database. It's important to keep track of them so that database users know what is available to them.

The goal of this exercise is to get familiar with viewing views within a database and interpreting their purpose. This is a skill needed when writing database documentation or organizing views.

```
-- Get all non-systems views
SELECT * FROM INFORMATION_SCHEMA.views
WHERE table_schema NOT IN ('pg_catalog', 'information_schema');
```

table_catalog	table_schema	table_name	view_definition	check_option	is_updatable	is_insertable_into	is_trigger_updat
dataarchpost	public	view1	SELECT content.reviewid, content.content FROM content WHERE (length(content.content) > 4000);	NONE	YES	YES	NO
dataarchpost	public	view2	SELECT reviews.reviewid, reviews.title, reviews.score FROM reviews WHERE (reviews.pub_year = 2017) ORDER BY reviews.score DESC LIMIT 10;	NONE	NO	NO	NO

What does `view1` do?

- ○ Returns the `content` records with `reviewid` s that have been viewed more than 4000 times.
- ◉ Returns the `content` records that have reviews of more than 4000 characters.
- ○ Returns the first 4000 records in `content` .

What does `view2` do?

- ○ Returns 10 random reviews published in 2017.
- ○ Returns the top 10 lowest scored reviews published in 2017.
- ◉ Returns the top 10 highest scored reviews published in 2017.

Note that unlike this exercise, you should always give views descriptive names for views!

Creating and querying a view

Have you ever found yourself running the same query over and over again? Maybe, you used to keep a text copy of the query in your desktop notes app, but that was all before you knew about views!

In these Pitchfork reviews, we're particularly interested in high-scoring reviews and if there's a common thread between the works that get high scores. In this exercise, you'll make a view to help with this analysis so that we don't have to type out the same query often to get these high-scoring reviews.

```
-- Create a view for reviews with a score above 9
CREATE VIEW high_scores AS
SELECT * FROM REVIEWS
WHERE score > 9;

-- Count the number of self-released works in high_scores
SELECT COUNT(*) FROM high_scores
INNER JOIN labels ON high_scores.reviewid = labels.reviewid
WHERE label = 'self-released';
```

count
3

Views are great because they're easy to set up and use immediately thereafter.

Managing views

Views can get as complicated and creative as you choose:

- **Aggregation:** `SUM()`, `AVG()`, `COUNT()`, `MIN()`, `MAX()`, `GROUP BY`, etc
- **Joins:** `INNER JOIN`, `LEFT JOIN`. `RIGHT JOIN`, `FULL JOIN`
- **Conditionals:** `WHERE`, `HAVING`, `UNIQUE`, `NOT NULL`, `AND`, `OR`, `>`, `<`, etc

Views are helpful for access control:

`GRANT privilege(s)` or `REVOKE privilege(s)`

`ON object`

`TO role` or `FROM role`

- **Privileges:** `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc
- **Objects:** table, view, schema, etc
- **Roles:** a database user or a group of database users

```
GRANT UPDATE ON ratings TO PUBLIC;
```

```
REVOKE INSERT ON films FROM db_user;
```

Updating a view

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Not all views are updatable

- View is made up of one table
- Doesn't use a window or aggregate function

Inserting into a view

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

Not all views are insertable

Takeaway: avoid modifying data through views

Generally, avoid modifying data through views. It's usually a good idea to use views for read-only purposes only.

Dropping a view

```
DROP VIEW view_name [ CASCADE | RESTRICT ];
```

- `RESTRICT` (default): returns an error if there are objects that depend on the view
- `CASCADE` : drops view and any object that depends on that view

Redefining a view

```
CREATE OR REPLACE VIEW view_name AS new_query
```

- If a view with `view_name` exists, it is replaced
- `new_query` must generate the same column names, order, and data types as the old query
- The column output may be different
- New columns may be added at the end

If these criteria can't be met, drop the existing view and create a new one

Altering a view

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO new_owner
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ]
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

Creating a view from other views

Views can be created from queries that include other views. This is useful when you have a complex schema, potentially due to normalization, because it helps reduce the `JOINS` needed. The biggest concern is keeping track of dependencies, specifically how any modifying or dropping of a view may affect other views.

In the next few exercises, we'll continue using the Pitchfork reviews data. There are two views of interest in this exercise. `top_15_2017` holds the top 15 highest scored reviews published in 2017 with columns `reviewid`,`title`, and `score`. `artist_title` returns a list of all reviewed titles and their respective artists with columns `reviewid`, `title`, and `artist`. From these views, we want to create a new view that gets the highest scoring artists of 2017.

```
-- Create a view with the top artists in 2017
CREATE VIEW top_artists_2017 AS
-- with only one column holding the artist field
SELECT artist_title.artist FROM artist_title
INNER JOIN top_15_2017
ON top_15_2017.reviewid = artist_title.reviewid;

-- Output the new view
SELECT * FROM top_artists_2017;
```

Which is the `DROP` command that would drop both `top_15_2017` and `top_artists_2017`?

- (●) `DROP VIEW top_15_2017 CASCADE;`
- ( ) `DROP VIEW top_15_2017 RESTRICT;`
- ( ) `DROP VIEW top_artists_2017 RESTRICT;`
- ( ) `DROP VIEW top_artists_2017 CASCADE;`

Because top_artists_2017 depends on top_15_2017, the CASCADE parameter indicates both should be dropped.

Granting and revoking access

Access control is a key aspect of database management. Not all database users have the same needs and goals, from analysts, clerks, data scientists, to data engineers. As a general rule of thumb, write access should never be the default and only be given when necessary.

In the case of our Pitchfork reviews, we don't want all database users to be able to write into the `long_reviews` view. Instead, the editor should be the only user able to edit this view.

```
-- Revoke everyone's update and insert privileges
REVOKE UPDATE, INSERT ON long_reviews FROM PUBLIC;

-- Grant editor update and insert privileges
GRANT UPDATE, INSERT ON long_reviews TO editor;
```

The editor user is now the only person who can UPDATE and INSERT on the long_reviews view.

Updatable views

In a previous exercise, we've used the `information_schema.views` to get all the views in a database. If you take a closer look at this table, you will notice a column that indicates whether the view is updatable.

```
SELECT * FROM INFORMATION_SCHEMA.views
WHERE table_schema NOT IN ('pg_catalog', 'information_schema');
```

table_catalog	table_schema	table_name	view_definition	check_option	is_updatable	is_insertable
dataarchpost	public	long_reviews	SELECT content.reviewid, content.content FROM content WHERE (length(content.content) > 4000);	NONE	YES	YES
dataarchpost	public	top_25_2017	SELECT reviews.reviewid, reviews.title, reviews.score FROM reviews WHERE (reviews.pub_year = 2017) ORDER BY reviews.score DESC LIMIT 25;	NONE	NO	NO
dataarchpost	public	artist_title	SELECT reviews.reviewid, reviews.title, artists.artist FROM (reviews JOIN artists ON ((artists.reviewid = reviews.reviewid)));	NONE	NO	NO

Which views are updatable?

- `long_reviews` and `top_25_2017`
- `top_25_2017`
- (selected) `long_reviews`
- `top_25_2017` and `artist_title`

Correct! long_reviews is updatable because it's made from one table and doesn't have any special clauses.

Redefining a view

Unlike inserting and updating, redefining a view doesn't mean modifying the actual data a view holds. Rather, it means modifying the underlying query that makes the view. In the last section, we learned of two ways to redefine a view: (1) `CREATE OR REPLACE` and (2) `DROP` then `CREATE`. `CREATE OR REPLACE` can only be used under certain conditions.

The `artist_title` view needs to be appended to include a column for the `label` field from the `labels` table.

Can the `CREATE OR REPLACE` statement be used to redefine the `artist_title` view?

- (selected) Yes, as long as the `label` column comes at the end.
- No, because the new query requires a `JOIN` with the `labels` table.
- No, because a new column that did not exist previously is being added to the view.
- Yes, as long as the `label` column has the same data type as the other columns in `artist_title`

Redefine the `artist_title` view to include a column for the `label` field from the `labels` table.

```sql
-- Redefine the artist_title view to have a label column
CREATE OR REPLACE VIEW artist_title AS
SELECT reviews.reviewid, reviews.title, artists.artist, labels.label
FROM reviews
INNER JOIN artists
ON artists.reviewid = reviews.reviewid
INNER JOIN labels
ON labels.reviewid = reviews.reviewid;

SELECT * FROM artist_title;
```

You redefined the artist_title succesfully using the CREATE OR REPLACE statement. If we had wanted to change the order of the columns completely, we would have had to drop the table and then create a new one using the same name.

Materialized views

Two types of views

Views

- Also known as **non-materialized views**
- ie, virtual

Materialized views

- Physically materialized

Instead of storing a query, a materialized view stores the query results. These query results are stored on disk. This means the query becomes precomputed via the view. When you query a materialized view, it accesses the stored query results on the disk, rather than running the query like a non-materialized view and creating a virtual table.
Materialized views are refreshed or rematerialized when prompted, ie, the query is run and the stored query results are updated. This can be scheduled depending on how often you expect the underlying query results are changing.

When to use materialized views

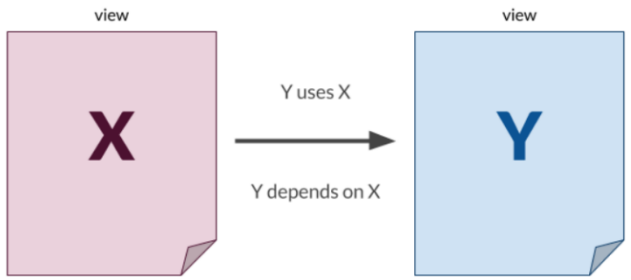
- Long running queries
- Underlying query results don't change often
- Data warehouses because OLAP is not write-intensive
  - Save on computational cost of frequent queries

Implementing materialized views

(in PostgreSQL)

```
CREATE MATERIALIZED VIEW my_mv AS SELECT * FROM existing_table;
```

```
REFRESH MATERIALIZED VIEW my_mv;
```



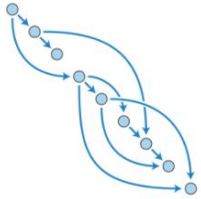
Unlike non-materialized views, you need to manage when you refresh materialized views when you have dependencies. For example, let's say you have two materialized views: X and Y. Y uses X in its query; meaning Y depends on X. X doesn't depend on Y as it doesn't use Y in its query. Let' s say X has a more time-consuming query. If Y is refreshed before X's refresh is completed, then Y now has out-of-date data.

Managing dependencies

- Materialized views often depend on other materialized views
- Creates a **dependency chain** when refreshing views
- Not the most efficient to refresh all views at the same time

Tools for managing dependencies

- Use Directed Acyclic Graphs (**DAGs**) to keep track of views



- Pipeline scheduler tools

Apache Airflow Luigi

Materialized versus non-materialized

Materialized and non-materialized are two distinct categories of views. In this exercise, you will organize their differences and similarities.

Non-Materialized Views	Non-Materialized & Materialized Views	Materialized Views
Always returns up-to-date data	Helps reduce the overhead of writing queries	Stores the query result on disk
Better to use on write-intensive databases	Can be used in a data warehouse	Consumes more storage

These characteristics can help make decisions about when to use a materialized or non-materialized view.

Creating and refreshing a materialized view

The syntax for creating materialized and non-materialized views are quite similar because they are both defined by a query. One key difference is that we can refresh materialized views, while no such concept exists for non-materialized views. It's important to know how to refresh a materialized view, otherwise the view will remain a snapshot of the time the view was created.

In this exercise, you will create a materialized view from the table `genres`. A new record will then be inserted into `genres`. To make sure the view has the latest data, it will have to be refreshed.

```sql
-- Create a materialized view called genre_count
CREATE MATERIALIZED VIEW genre_count AS
SELECT genre, COUNT(*)
FROM genres
GROUP BY genre;

INSERT INTO genres
VALUES (50000, 'classical');

-- Refresh genre_count
REFRESH MATERIALIZED VIEW genre_count;

SELECT * FROM genre_count;
```

If we didn't include that REFRESH statement, genre_count would not have a row for the classical genre since that genre did not exist before our INSERT statement.

Managing materialized views

Why do companies use pipeline schedulers, such as Airflow and Luigi, to manage materialized views?

- ○ To set up a data warehouse and make sure tables have the most up-to-date data.
- ◉ To refresh materialized views with consideration to dependences between views.
- ○ To convert non-materialized views to materialized views.
- ○ To prevent the creation of new materialized views when there are too many dependencies.

These pipeline schedulers help visualize dependencies and create a logical order for refreshing views.

Chapter 4. Database Management

This final chapter ends with some database management-related topics. You will learn how to grant database access based on user roles, how to partition tables into smaller pieces, what to keep in mind when integrating data, and which DBMS fits your business needs best.

Database roles and access control

Granting and revoking access to a view

`GRANT privilege(s)` or `REVOKE privilege(s)`

`ON object`

`TO role` or `FROM role`

- **Privileges:** `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc.
- **Objects:** table, view, schema, etc.
- **Roles:** a database user or a group of database users

```
GRANT UPDATE ON ratings TO PUBLIC;
```

```
REVOKE INSERT ON films FROM db_user;
```

Database roles

- Manage database access permissions
- A database role is an entity that contains information that:
  - Define the role's privileges
    - Can you login?
    - Can you create databases?
    - Can you write to tables?
  - Interact with the client authentication system
    - Password
- Roles can be assigned to one or more users
- Roles are global across a database cluster installation

Create a role

- Empty role

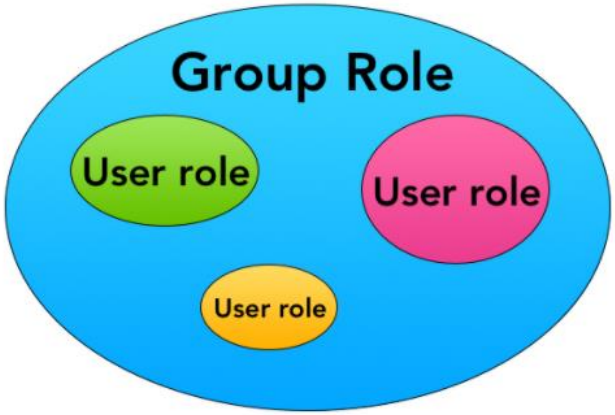
```
CREATE ROLE data_analyst;
```

- Roles with some attributes set

```
CREATE ROLE intern WITH PASSWORD 'PasswordForIntern' VALID UNTIL '2020-01-01';
```

```
CREATE ROLE admin CREATEDB;
```

```
ALTER ROLE admin CREATEROLE;
```



GRANT and REVOKE privileges from roles

```
GRANT UPDATE ON ratings TO data_analyst;
```

```
REVOKE UPDATE ON ratings FROM data_analyst;
```

The available privileges in PostgreSQL are:

- `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE`

Group role

```
CREATE ROLE data_analyst;
```

User role

```
CREATE ROLE intern WITH PASSWORD 'PasswordForIntern' VALID UNTIL '2020-01-01';
```

```
GRANT data_analyst TO alex;
```

```
REVOKE data_analyst FROM alex;
```

Common PostgreSQL roles

Role	Allowed access
pg_read_all_settings	Read all configuration variables, even those normally visible only to superusers.
pg_read_all_stats	Read all pg_stat_* views and use various statistics related extensions, even those normally visible only to superusers.
pg_signal_backend	Send signals to other backends (eg: cancel query, terminate).
More...	More...

Benefits and pitfalls of roles

Benefits

- Roles live on after users are deleted
- Roles can be created before user accounts
- Save DBAs time

Pitfalls

- Sometimes a role gives a specific user too much access
  - You need to pay attention

Create a role

A database role is an entity that contains information that define the role's privileges and interact with the client authentication system. Roles allow you to give different people (and often groups of people) that interact with your data different levels of access.

Imagine you founded a startup. You are about to hire a group of data scientists. You also hired someone named Marta who needs to be able to login to your database. You're also about to hire a database administrator. In this exercise, you will create these roles.

```
-- Create a data scientist role
CREATE ROLE data_scientist;
```

```
-- Create a role for Marta
CREATE ROLE marta LOGIN;
```

```
-- Create an admin role
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

You created a group role, data_scientist, that you can populate later with whatever access level you deem appropriate. Marta can login. The admin, whoever holds that role, has the ability to create databases and manage roles. You now know how to create roles to specify different levels of access for individuals and groups of individuals, which is good database management practice.

GRANT privileges and ALTER attributes

Once roles are created, you grant them specific access control privileges on objects, like tables and views. Common privileges being `SELECT`, `INSERT`, `UPDATE`, etc.

Imagine you're a cofounder of that startup and you want all of your data scientists to be able to update and insert data in the `long_reviews` view. In this exercise, you will enable those soon-to-be-hired data scientists by granting their role (`data_scientist`) those privileges. Also, you'll give Marta's role a password.

```
-- Grant data_scientist update and insert privileges
GRANT UPDATE, INSERT ON long_reviews TO data_scientist;

-- Give Marta's role a password
ALTER ROLE marta WITH PASSWORD 's3cur3p@ssw0rd';
```

Everyone in the data_scientist role (which is currently no one, though you're hiring shortly) is now able to update data and insert data in the long_reviews view. This view has business-critical data that's updated often so these privileges are key to your startup's success. Marta is happy because she has a password now, too!

Add a user role to a group role

There are two types of roles: user roles and group roles. By assigning a user role to a group role, a database administrator can add complicated levels of access to their databases with one simple command.

For your startup, your search for data scientist hires is taking longer than expected. Fortunately, it turns out that Marta, your recent hire, has previous data science experience and she's willing to chip in the interim. In this exercise, you'll add Marta's user role to the data scientist group role. You'll then remove her after you complete your hiring process.

```
-- Add Marta to the data scientist group
GRANT data_scientist TO marta;

-- Celebrate! You hired data scientists.

-- Remove Marta from the data scientist group
REVOKE data_scientist FROM marta;
```

Bless you, Marta! She really helped the company out in a pinch. And it wasn't difficult for you to set her up with appropriate access to company data thanks to the roles you previously created!

Table partitioning

Why partition?

Tables grow (100s Gb / Tb)

Problem: queries/updates become slower

Because: e.g., indices don't fit memory

Solution: split table into smaller parts (= **partitioning**)

The data you'll access or update is still the same. The difference is we distribute the data over several physical entities.

Vertical partitioning: an example

id	name	short_description	price	long_description
1	Sunglasses	Cool sunglasses	$25	When you put on these ...
2	Leather wallet	Leather wallet with writing	$127	The wallet says 'Bad ...

id	name	short_description	price
1	Sunglasses	Cool sunglasses	$25
2	Leather wallet	Leather wallet with writing	$127

id	long_description
1	When you put on these ...
2	The wallet says 'Bad ...

E.g., store `long_description` on slower medium

Horizontal partitioning

Key	Column 1	timestamp
1	...	2018-01-01
2	...	...
3	...	2018-31-01
4	...	2019-01-01
5	...	...
6	...	2019-31-01

Key	Column 1	timestamp
1	...	2018-01-01
2	...	...
3	...	2018-31-01

Key	Column 1	timestamp
4	...	2019-01-01
5	...	...
6	...	2019-31-01

id	product_id	amount	total_price	timestamp	
1	123	1	$102	2019-04-01	Q1 partition
2	101	7	$21	2019-23-02	
3	18202	1	$499	2019-30-04	Q2 partition
4	1762	15	$1500	2019-21-08	Q3 partition
5	10	1	$5	2019-30-08	
6	123	1	$102	2019-29-09	Q4 partition

```
CREATE TABLE sales (
    ...
    timestamp DATE NOT NULL
)
PARTITION BY RANGE (timestamp);

CREATE TABLE sales_2019_q1 PARTITION OF sales
    FOR VALUES FROM ('2019-01-01') TO ('2019-03-31');
...
CREATE TABLE sales_2019_q4 PARTITION OF sales
    FOR VALUES FROM ('2019-09-01') TO ('2019-12-31');
CREATE INDEX ON sales ('timestamp');
```

Pros/cons of horizontal partitioning

Pros

- Indices of **heavily-used partitions** fit in memory
- Move to **specific medium**: slower vs. faster
- Used for both OLAP as OLTP

Cons

- Partitioning **existing table** can be a hassle
- Some **constraints** can not be set

We can take partitioning one step further and distribute the partitions over several machines. When horizontal partitioning is applied to spread a table over several machines, it's called **sharding**. You can see how this relates to massively parallel processing databases, where each node, or machine, can do calculations on specific shards.

Reasons to partition

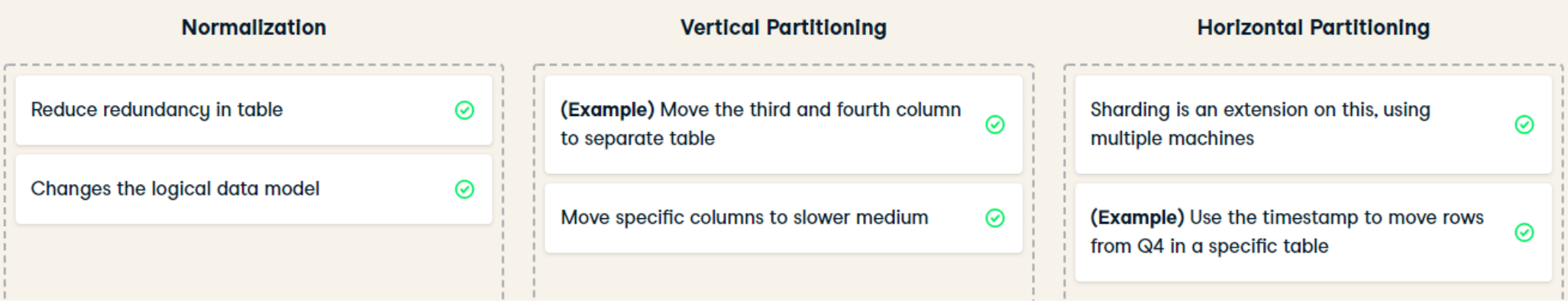
In the previous section, you saw some very good reasons to use partitioning. However, can you find which one wouldn't be a good reason to use partitioning?

- (●) Improve data integrity
- ( ) Save records from 2017 or earlier on a slower medium
- ( ) Easily extend partitioning to sharding, and thus making use of parallelization

Partitioning and normalization

In the previous section, you saw the differences between the two types of partitioning: **vertical** and **horizontal** partitioning. As you'd expect, the names suggest how these different strategies work.

It might be a bit challenging to distinguish **normalization**, which you saw in previous chapters, from **partitioning**.



Partitioning is related to the physical data model. It does not change the logical data model, while normalization does.

Creating vertical partitions

For vertical partitioning, there is no specific syntax in PostgreSQL. You have to create a new table with particular columns and copy the data there. Afterward, you can drop the columns you want in the separate partition. If you need to access the full table, you can do so by using a `JOIN` clause.

In this exercise and the next one, you'll be working with the example database called `pagila`. It's a database that is often used to showcase PostgreSQL features. The database contains several tables. We'll be working with the `film` table. In this exercise, we'll use the following columns:

- `film_id`: the unique identifier of the film
- `long_description`: a lengthy description of the film

```
-- Create a new table called film_descriptions
CREATE TABLE film_descriptions (
    film_id INT,
    long_description TEXT
);

-- Copy the descriptions from the film table
INSERT INTO film_descriptions
SELECT film_id, long_description FROM film;

-- Drop the column in the original table
ALTER TABLE film DROP COLUMN long_description;

-- Join to create the original table
SELECT * FROM film
JOIN film_descriptions USING(film_id);
```

Now you know how to CREATE, INSERT and ALTER statements!

Creating horizontal partitions

In the previous section, you also learned about horizontal partitioning.

The example of horizontal partitioning showed the syntax necessary to create horizontal partitions in PostgreSQL. If you need a reminder, you can have a look at the slides.

In this exercise, however, you'll be using a list partition instead of a range partition. For list partitions, you form partitions by checking whether the partition key is in a list of values or not.

To do this, we partition by `LIST` instead of `RANGE`. When creating the partitions, you should check if the values are `IN` a list of values.

We'll be using the following columns in this exercise:

- `film_id`: the unique identifier of the film
- `title`: the title of the film
- `release_year`: the year it's released

```
-- Create a new table called film_partitioned
CREATE TABLE film_partitioned (
  film_id INT,
  title TEXT NOT NULL,
  release_year TEXT
)
PARTITION BY LIST (release_year);

-- Create the partitions for 2019, 2018, and 2017
CREATE TABLE film_2019
  PARTITION OF film_partitioned FOR VALUES IN ('2019');

CREATE TABLE film_2018
  PARTITION OF film_partitioned FOR VALUES IN ('2018');

CREATE TABLE film_2017
  PARTITION OF film_partitioned FOR VALUES IN ('2017');

-- Insert the data into film_partitioned
INSERT INTO film_partitioned
SELECT film_id, title, release_year FROM film;
```

```
-- View film_partitioned
SELECT * FROM film_partitioned;
```

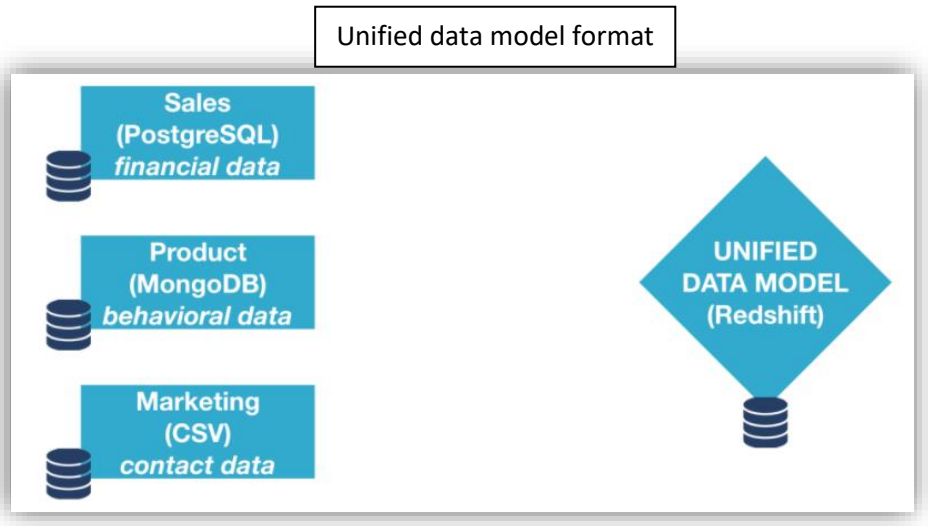
As you can see, the data is not changed in the partitioned table. However, you might notice PostgreSQL orders the partitioned table differently by default.

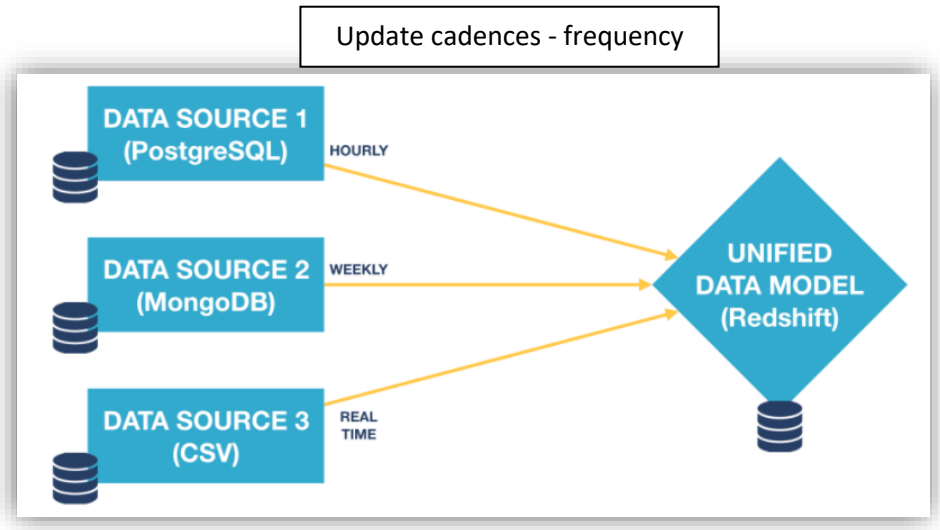
Data integration

Data Integration combines data from different sources, formats, technologies to provide users with a translated and unified view of that data.

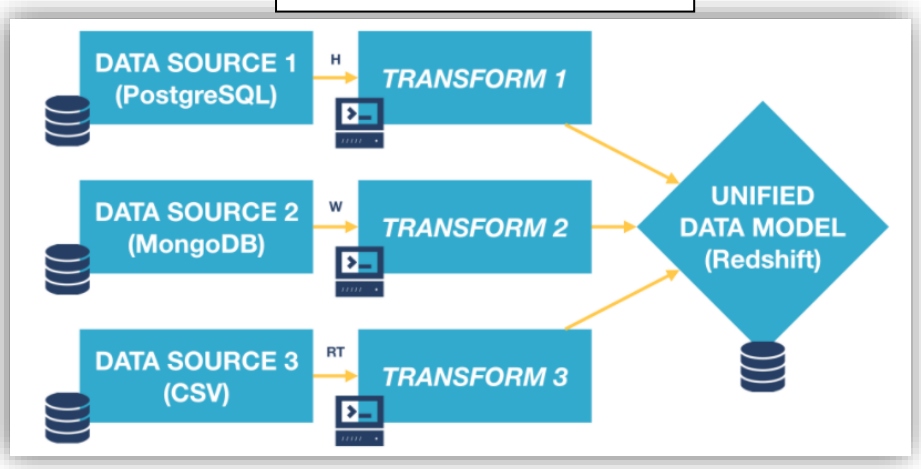
Business case examples

- 360-degree customer view
- Acquisition
- Legacy systems

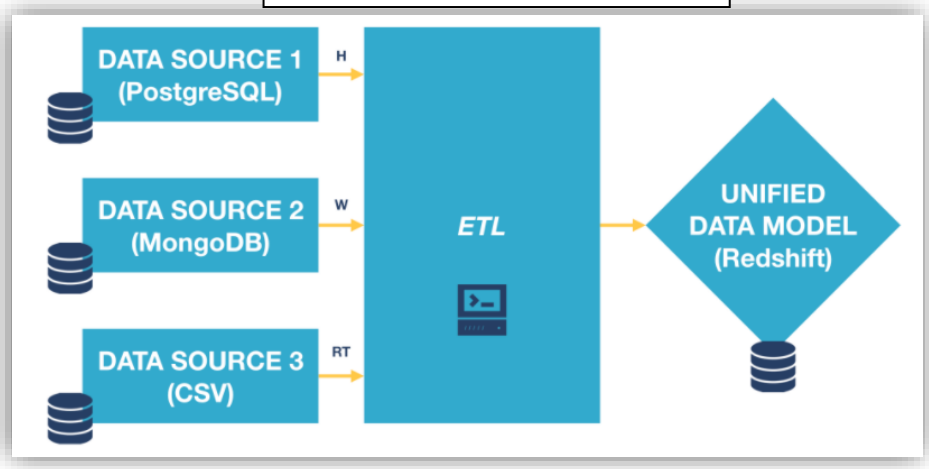




Data transformation

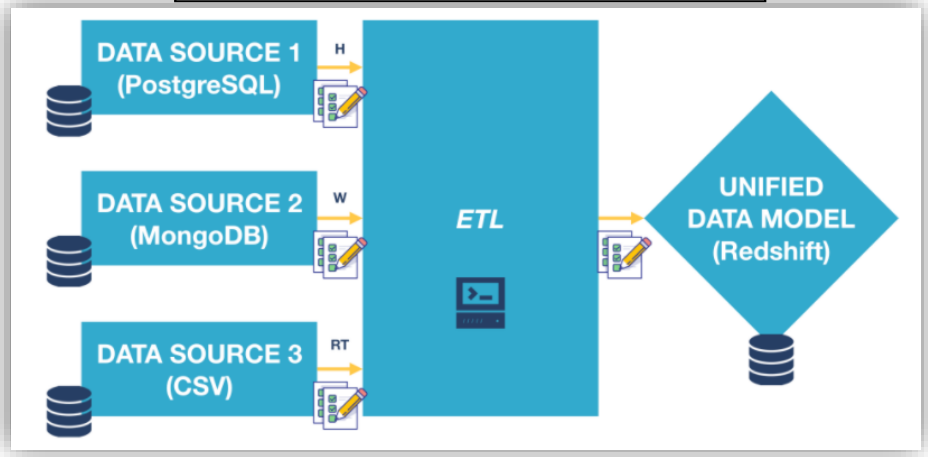


ETL tools, eg. Airflow or Scriptella

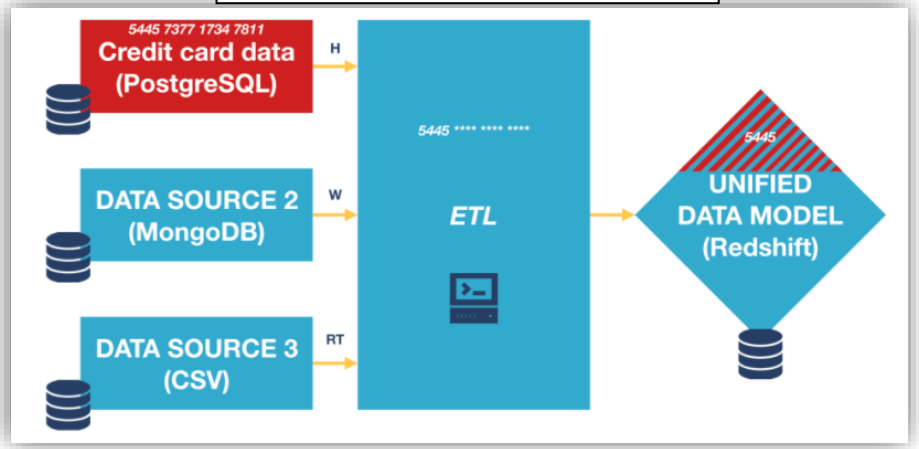


When choosing your tool, you must ensure that it's flexible enough to connect to all of your data sources. Reliable, so that it can still be maintained in a year. And it should scale well, anticipating an increase in data volume and sources.

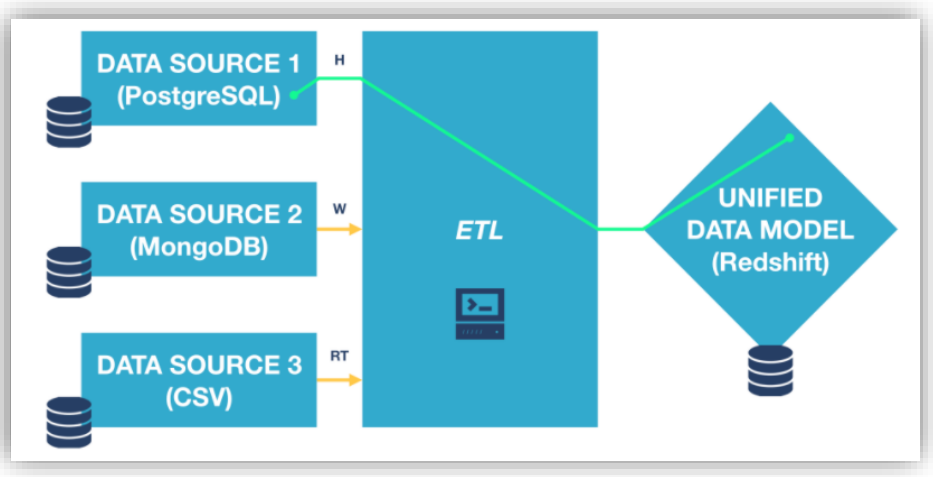
Automated testing and proactive alerts



Security - credit card anonymization



Data governance - lineage



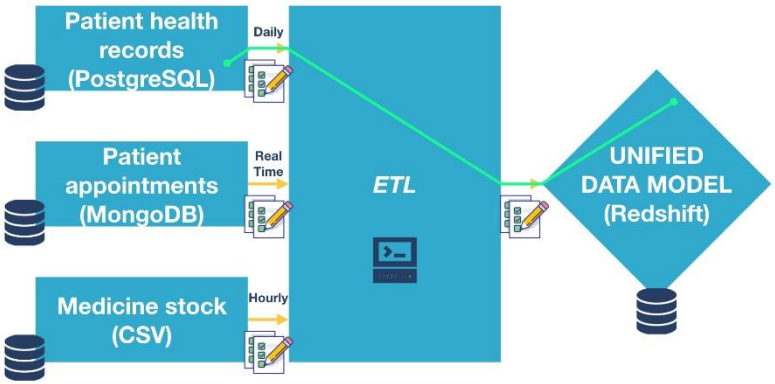
Data integration do's and dont's

You just learned a lot about data integration, let's check your understanding of the concepts.

False	True
All your data has to be updated in real time in the final view.	Data integration should be business driven, e.g. what combination of data will be useful for the business.
Automated testing and proactive alerts are not needed.	Being able to access the desired data through a single view does not mean all data is stored together.
Everybody should have access to sensitive data in the final view.	My source data can be stored in different physical locations.
Your data integration solution, hand-coded or ETL tool, should work once and then you can use the resulting view to run queries forever.	My source data can be in different formats and database management systems.
After data integration all your data should be in a single table.	Data in the final view can be updated in different intervals.
You should choose whichever solution is right for the job right now.	You should be careful choosing a hand-coded solution because of maintenance cost.

Analyzing a data integration plan

You're a data analyst in a hospital that wants to make sure there is enough cough medicine should an epidemic break out. For this, you need to combine the historical health records with the upcoming appointments to see if you can detect a pattern similar to the last cold epidemic. Then, you need to make sure there is sufficient stock available or if the stock should be increased. To help tackle this problem, you created a data integration plan.



Which risk is not clearly indicated on the data integration plan?

- It is unclear if you took data governance into account.
- You didn't clearly show where your data originated from.
- **You should indicate that you plan to anonymize patient health records.**
- If data is lost during ETL you will not find out.

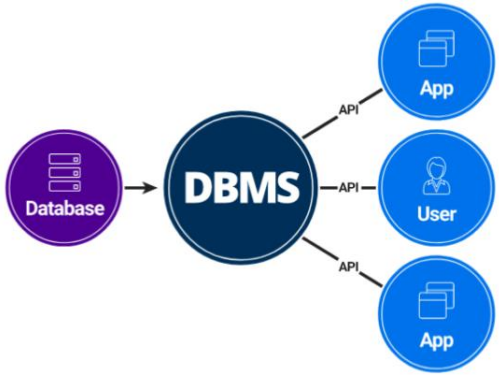
When working with sensitive data it is important to think about permissions. By default you should have the same access rights before and after data integration. If part of the data is essential, it should be anonymized, in this case you can keep the illnesses but remove identifying information.

Picking a Database Management System (DBMS)

A DBMS is a system software for creating and maintaining databases.
The DBMS manages three important aspects:

- the data,
- the database schema which defines the database's logical structure, and
- the database engine that allows data to be accessed, locked and modified.

Essentially, the DBMS serves as an interface between the database and end users or application programs.



SQL DBMS

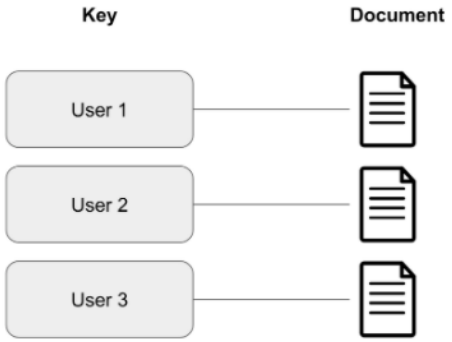
- Relational DataBase Management System (**RDBMS**)
- Based on the relational model of data
- Query language: SQL
- Best option when:
  - Data is structured and unchanging
  - Data must be consistent

NoSQL DBMS

- Less structured
- Document-centered rather than table-centered
- Data doesn't have to fit into well-defined rows and columns
- Best option when:
  - Rapid growth
  - No clear schema definitions
  - Large quantities of data
- Types: key-value store, document store, columnar database, graph database

- Combinations of keys and values
  - Key: unique identifier
  - Value: anything
- **Use case**: managing the shopping cart for an on-line buyer

- Example:

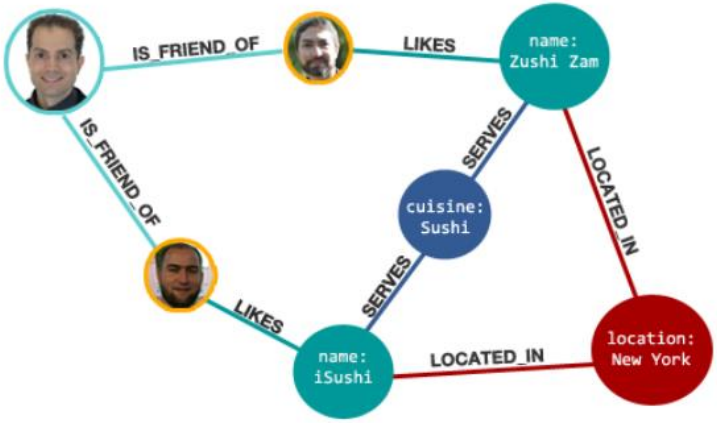


- Similar to key-value
- Values (= documents) are structured
- **Use case:** content management
- Example:

mongoDB.

- Store data in columns
- Scalable
- **Use case:** big data analytics where speed is important
- Example:

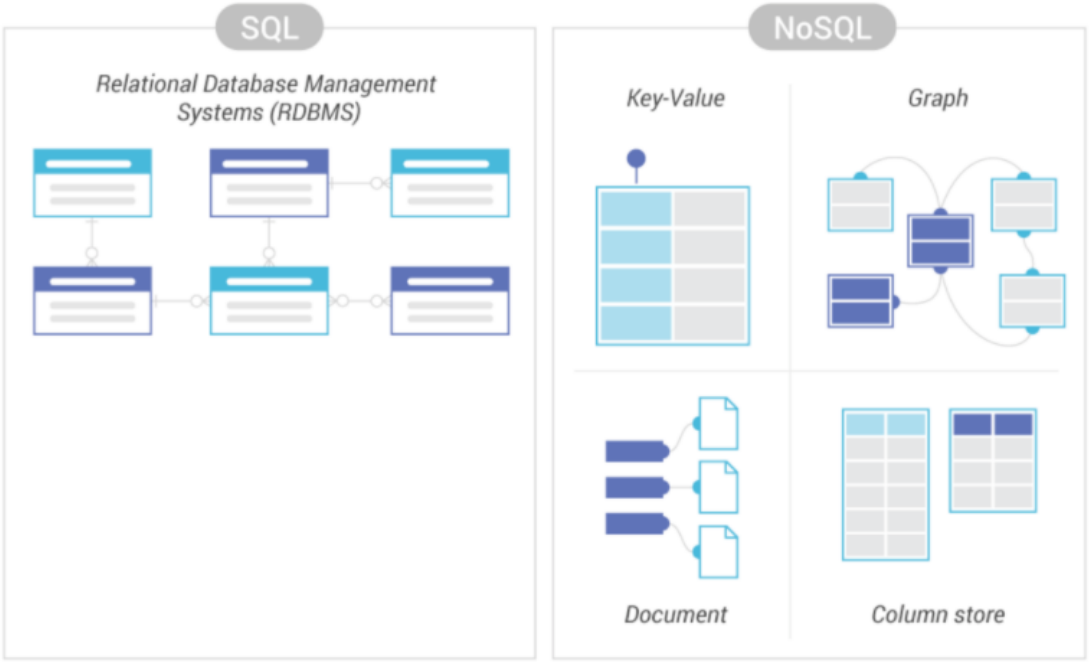
cassandra



- Data is interconnected and best represented as a graph
- **Use case:** social media data, recommendations
- Example:

neo4j

Choosing a DBMS



If your application has a fixed structure and doesn't need frequent modifications, a SQL DBMS is preferable.

Conversely, if you have applications where data is changing frequently and growing rapidly, like in big data analytics, NoSQL is the best option for you.

SQL versus NoSQL

Deciding when to use a SQL versus NoSQL DBMS depends on the kind of information you're storing and the best way to store it. Both types store data, they just store data differently. When is it better to use a SQL DBMS?

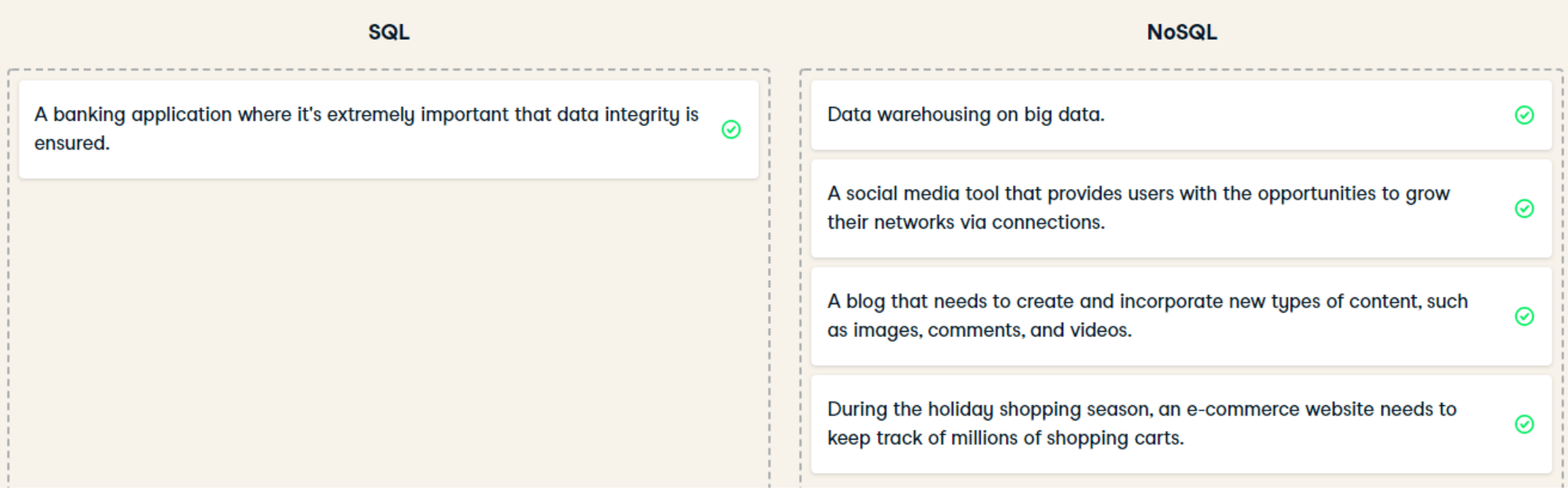
- ○ You are dealing with rapidly evolving features, functions, data types, and it's difficult to predict how the application will grow over time.
- ○ You have a lot of data, many different data types, and your data needs will only grow over time.
- ◉ You are concerned about data consistency and 100% data integrity is your top goal.
- ○ Your data needs scale up, out, and down.

The strength of SQL DBMSs lies in using integrity constraints to maintain data consistency across multiple tables.

Choosing the right DBMS

As you saw in the previous section, there are lots of different options when choosing a DBMS. The choice depends on the business need. In this exercise, you are given a list of cards describing different scenarios and it's your job to pick the DBMS type that fits the project best. Remember the different DBMS types:

- **SQL**: RDBMS
- **NoSQL**: key-value store, document store, columnar database, graph database



As you can see there are many different DBMS types and you need to carefully consider the business needs before making your decision.

Course completed!

Recap topics covered:

- OLTP and OLAP
- Storing data and storage solutions
- Data types and data models
- ETL and ELT
- Database design
- Difference between fact and dimension tables
- Star schema and Snowflake schema
- Primary keys and foreign keys
- Querying the star/snowflake schema
- Normal forms: 1NF, 2NF, 3NF
- Database views vs tables
- Managing views
- Materialized views
- Database roles and access control
- Partitioning and normalization
- Vertical and horizontal partition
- Data integration
- SQL and NoSQL
- DBMS

Happy learning!