**Wstęp do baz danych i SQLite**

Nie bez przyczyny poświęcamy oddzielny moduł szkoleniowy na opanowanie SQLa i ogólnie pojęcia relacyjnych baz danych. SQL jest bowiem jednym z najczęściej wykorzystywanych języków w pracy programisty. Może być on w pełni zintergrowany z choćby takim językiem programowania jak Python i służyć do zarządzania dużymi zbiorami danych.

Zauważ, że bazy danych wykorzystywane są w zasadzie we wszystkich aplikacjach codziennego użytku (konieczność posługiwania się systemem przechowującym dane o użytkownikach), a także szczególnie w pracy jako web developer czy data analyst.

Ponadto własne praktyczne aplikacje, które możemy spieniężyć, nie obejdą się bez interakcji z przechowywanymi danymi.

**Bazy danych i ich język**

Bazy danych to nic innego jak miejsca, w których przechowywane są różne informacje potrzebne do prawidłowego funkcjonowania różnych systemów (mogą to być informacje z firmy, aplikacji, urządzeń).

Przechowywanie informacji to jedno, ale dodatkową zaletą funkcjonowania baz danych jest to, że możemy łatwo przetwarzać i analizować takie informacje. Za pomocą języka, który obsługuje dane, możemy tworzyć, czytać, modyfikować i usuwać informacje (tzw. 4 podstawowe funkcje **CRUD - Create, Read, Update, Delete**).

W zależności od rodziny, bazy danych możemy podzielić na: **relacyjne** lub **nierelacyjne**.

Z pierwszym rodzajem baz identyfikujemy skrót **SQL** *Structured Query Language*(omawiany w kursie), a z nierelacyjnymi - **NoSQL** (Non Relational Structured Query Language).

Rodzaje te różnią się mechanizmem przechowywania i wyszukiwania danych modelowanych (brzmi strasznie, ale chodzi przede wszystkim o to, że w NoSQL nie ma bezpośrednich powiązań - relacji - między zbiorami danych).

Co ciekawe, koncepcja nierelacyjnych baz danych nie jest nowa, a wykorzystanie nosql-owych repozytoriów rozpoczęło się w czasach pierwszych komputerów.

A to dlatego, że mechanizm NoSQL jest szybszy i często postrzegany jako bardziej elastyczny niż SQL. Nie jest jednak tak, że jeden rodzaj jest lepszy od drugiego i powinniśmy ograniczać się tylko do jednego rozwiązania.

Wybór konkretnej bazy danych pod względem formy przechowywania danych zależy od wielu czynników, w tym problemu, który musi ona rozwiązać.

Wiemy już, że wyróżniamy 2 rodzaje baz danych, ale co ciekawe, każdą z tych grup możemy podzielić na różne implementacje. Z racji tego, że kurs ten ma wdrożyć Cię do relacyjnych baz danych, skupimy się na implementacjach właśnie tego rodzaju baz, a są to m.in:

* PostgreSQL
* Oracle
* MySQL
* SQL Server
* **SQLite**

Nie przejmuj się, nie musisz być zapoznany z każdą implementacją.

Okazuje się bowiem, że są one do siebie mocno zbliżone składniowo i występują tylko drobne różnice między nimi.

Tak więc, wystarczy samo opanowanie języka SQL, a przywyknięcie do pracy w różnych środowiskach wykorzystujących ten język, będzie czystą formalnością.

Zauważ jeszcze, że implementacja **SQLite** została przez Nas wyszczególniona w powyższym zestawieniu. To dlatego, że właśnie z niej będziemy korzystali w dalszej części kursu.

Ale po kolei, najpierw dogłębnie poznajmy wszystkie zasady i cały system pracyz bazami danych.

**Zasada działania relacyjnych baz danych**

Aby móc pracować na danych w bazie, konieczne jest nawiązanie połączenia z serwerem, który będzie obsługiwał żądania użytkownika i przeprowadzał odpowiednie operacje na bazie.

Komunikacja między klientem a serwerem odbywa się na określonym porcie (wyobrażaj to sobie jako drzwi klienta do różnych usług internetowych), np. PostgreSQL domyślnie używa portu 5432.

Ponadto SQLite, w celu komunikacji z bazą może wykorzystywać protokół **ODBC (Open Database Connectivity)** do obsługiwania klienckich żądań.

Do obsługi tych żądań wykorzystuje się sterowniki, których implementację zapewnia producent. To właśnie bezpośrednio pracując na tych sterownikach, klienci (czyli my - użytkownicy) mogą wysyłać różne polecenia do bazy przy użyciu języka SQL.

Serwer interpretuje te zapytania, wykonuje różne operacje (które mogą być jednym z zbioru funkcjonalności **CRUD**), modyfikuje dane i zwraca odpowiedni wynik. W ten sposób możliwe jest efektywne i w razie potrzeby zautomatyzowane (pewne czynności możemy zaprogramować) przetwarzanie dużych zbiorów danych.

**Pojęcia związane z relacyjnym modelem baz danych**

Po zapoznaniu się z tym, jak realizowana jest obsługa zapytań **(tzw. queries)** i komunikacja z bazą danych, nadszedł czas na przejście do nieco praktyczniejszych zagadnień.

Otóż w pracy z bazami, tak naprawdę wykorzystujesz **tabele (encje), atrybuty, relacje, powiązania, krotki, zapytania, klucze**.

Wyobraź sobie bazę danych firmy informatycznej. Firma ta przechowuje informacje o dostępnym **towarze w magazynie, producencie, oferowanych usługach, klientach firmy, pracownikach.**

**Tabele i atrybuty**

Do reprezentacji powyższych obiektów (towar, producent, usługa, firma itd.) służą właśnie **tabele**, nazywane też **encjami**.

Ponadto każda z nich przechowuje **atrybuty**. Atrybutami dla tabeli towar może być: nazwa towaru, ilość dostępna na magazynie, cena za jednostkę, ale również bardziej nietypowe cechy jak: **id towaru (unikalny)**, producent towaru.

Jeżeli znasz któryś z obiektowych języków programowania, to taka hermetyzacja i ułożenie danych powinno przypominać Ci klasy i ich pola.

I tu się z Tobą zgodzę! Encje to nic innego właśnie struktury przypominające klasy, które wykorzystywaliśmy choćby przy programowaniu w Pythonie.

Jak już się zapewne domyślasz, atrybutem encji może być dowolny typ danych, np. wartość logiczna, liczba całkowita, tekst.

**Krotki**

Krotki są ściśle związane z tabelami. Krotka to po prostu przechowywany obiekt w tabeli. Często są one reprezentowane w postaci wiersza. Każdy taki wiersz posiada również atrybuty, np. poniżej przedstawiono **dwie krotki o atrybutach**: Imię, Nazwisko, Wiek.

| Imię | Nazwisko | Wiek |
| --- | --- | --- |
| Kamil | Janczewski | 18 |
| Krzysztof | Bomba | 21 |

**Klucz główny**

Wyobraź sobie, że pewna tabela przechowuje krotki reprezentujące pracowników firmy. Aby móc efektywnie przydzielać zadania danym pracownikom, kontrolować ich czas pracy, konieczne jest przydzielenie każdemu **z nich unikalnego identyfikatora**.

Czyli czegoś, co będzie dla pojedynczego pracownika niepowtarzalne i nie będzie z nikim/niczym innym, w obrębie danej tabeli, skojarzone.

W odniesieniu do relacyjnych baz danych, ten unikalny element to tzw. **klucz główny (primary key, PK)**.

***Tabela pracownicy z kluczem głównym ID***

| ID | Imię | Nazwisko | Wiek | Staż pracy (lata) | Stanowisko |
| --- | --- | --- | --- | --- | --- |
| 0 | Kamil | Janczewski | 18 | 2.0 | Manager |
| 1 | Krzysztof | Bomba | 21 | 0.5 | Dozorca |

Klucze główne ułatwiają i przyspieszają pracę, ponadto niwelują dwuznaczność danych. Wyobraź sobie sytuację, że w bazie przechowujesz dwóch pracowników o imieniu Jan i nazwisku Kowalski. W systemie konieczne wówczas by było rozróżnianie tych dwóch osób.

Aby to zrealizować, konieczne właśnie jest dodanie unikalnego klucza w postaci np. **ID** lub numeru PESEL.

Klucze główne (przekształcone w klucze obce) ponadto umożliwiają tworzenie powiązań i skojarzeń pomiędzy tabelami. Więcej za chwilę.

**Klucz obcy**

Klucze obce (foreign keys, FKs) to ważne pojęcie pojawiające się przy tworzeniu zależności między tabelami. Chcąc stworzyć prostą definicję tego zagadnienia, moglibyśmy przytoczyć coś takiego: "Klucze obce to po prostu klucze główne danych krotek z określonej tabeli, **umieszczone w obcej tabeli**, np. odwołania do działów, do których jest przypisany dany pracownik”

Tak więc powiązanie między tabelą Pracownicy a Działy moglibyśmy zrealizować tak jak poniżej:

***Tabela Działy***

| ID | Nazwa | Kierownik |
| --- | --- | --- |
| 1 | HR | Bogusław L. |
| 2 | R&D | Anna J. |
| ... | ... | ... |

***Tabela Pracownicy***

| ID | Imię | Nazwisko | Wiek | Staż pracy (lata) | Stanowisko | Dział (FK) |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | Kamil | Janczewski | 18 | 2.0 | Manager | 1 |
| 1 | Krzysztof | Bomba | 21 | 0.5 | Dozorca | 2 |

W tabeli Pracownicy, odwołaliśmy się do danych działów z Działy poprzez umieszczenie klucza głównego (ID) danego działu w tabeli Pracownicy w kolumnie Dział (FK).

Tak więc możemy odczytać, że pracownik Kamil Janczewski jest przypisany do działu HR, którego kierownikiem jest Bogusław L. (nie to nie Bogusław Linda :)) a Krzysztof Bomba do R&D, który nadzoruje natomiast Anna J.

Zapytasz więc, dlaczego rozdzielać pracowników i stanowiska na dwie tabele, skoro można by było dodać do tabeli Pracownicy atrybuty takie jak: **Nazwa\_Działu**, **Kierownik\_Działu** i niepotrzebne byłoby rozbijanie danych na dwie tabele.

Faktycznie rozwiązanie to mogłoby zadziałać, ale okazałoby się całkowicie nieefektywne. Tabela Pracownicy byłaby przeładowana kolumnami, a struktura projektu ciężka do modyfikacji... Lepiej jest więc grupować określone dane w różnych tabelach i łączyć je ze sobą za pomocą powiązań.

**Rodzaje powiązań:**

Zobaczyłeś już, jak powiązać ze sobą dwie tabele danych. Dokładniej - zastosowaliśmy wiązanie jeden do jednego. Powinieneś być jednak świadomy, że nie jest to jedyny sposób na łączenie danych...

* **Jeden do jednego**

Jest to typ powiązania, który możemy wyobrażać sobie np. jako: **człowiek - pesel**. Zauważ bowiem, że z jednym człowiekiem możemy identyfikować tylko i wyłącznie jeden numer PESEL i na odwrót - z jednym numerem PESEL tylko jednego człowieka. Powiązanie jeden do jeden można określać jako 1 : 1.

* **Jeden do wielu**

Idąc dalej, ale kierując się podobną analogią, co w poprzednim przykładzie - w takiej sytuacji jedna osoba może być powiązana z kilkoma samochodami i vice versa. Jeden do wielu to inaczej 1 : n.

* **Wiele do wielu**

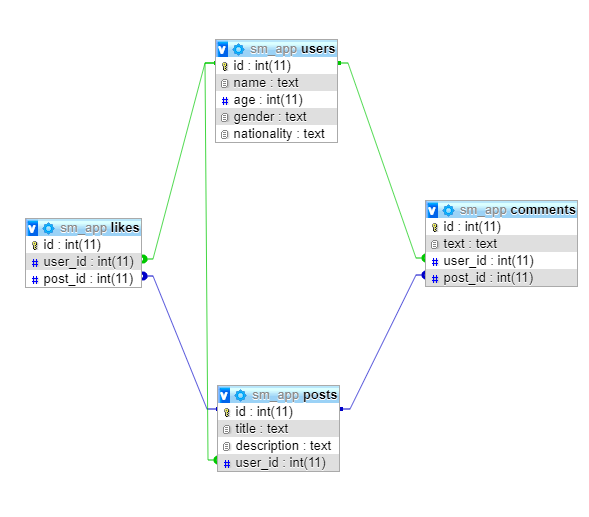
I ostatnim przykładem powiązania n : m może być przykład z różnymi produktami dostarczanymi przez kilku producentów. Na przykład producentami zestawu karty graficznej i laptopa może być firma ASUS oraz MSI. Tak więc z pojedynczym towarem (np. kartą graficzną) możemy skojarzyć wymienione firmy, a z pojedyncznym producentem (np. ASUS) kilka produktów (karta graficzna i laptop).

**Praca z diagramami baz danych**

Teorię już z grubsza poznaliśmy, zatem nadszedł czas na zapoznanie się z diagramami. Rzecz jasna, na ścieżce swojej pracy z bazami danych, poznasz wiele sposobów na graficzną reprezentacją danych.

Sposoby prezentowania danych są jednak w dużej mierze do siebie zbliżone, także warto zaznajomić się z samą ideą.

***Przykładowa baza z 4 tabelami i powiązaniami 1 : n.***



Zgodnie z tym, co ukazuje diagram - mamy do dyspozycji 4 tabele o nazwach: **users**, **likes**, **posts**, **comments**.

Każda z tych tabel przechowuje atrybuty takie jak, np.

1. Tabela posts: id (PK), title, description, user\_id (FK z tabeli users)
2. Tabela like: id (PK), user\_id (FK z tabeli users), post\_id (FK z tabeli posts)
3. etc.

Ważny jest zastosowany tutaj rodzaj powiązań między tabelami. Należy on do grupy "**jeden do wielu**" (zauważ, jak wyglądają końcówki zielonych i niebieskich linii reprezentujących zależności między tabelami).

Tak więc z jednym użytkownikiem możemy **skojarzyć wiele komentarzy**, ale jeden **konkretny komentarz tylko z jednym użytkownikiem**. Pojedynczy post z wieloma "lajkami", ale jeden konkretny "lajk" tylko z jednym postem oraz użytkownikiem itd.

**Operacje w języku SQL**

W języku SQL możemy wyróżnić kilka rodzajów operacji przetwarzania danych.   
Są to:

* **DML - Data Manipulation Language** - instrukcje manipulujące danymi, np. **usuwanie**, **dodawanie krotek**, **modyfikacja zawartości.**
* **DDL** **- Data Definition Language** - służą do manipulowania strukturą danych, np. dodawanie/usuwanie atrybutów, zmiana typu
* **DCL - Data Control Language** - służą do sterowania uprawnieniami grup/użytkowników na serwerze.
* **DQL - Data Query Language** - polecenia tego pokroju umożliwiają odczytywanie danych z bazy

**DML - Data Manipulation Language**

Najczęściej stosowanymi poleceniami wykonywanymi w ramach tej grupy operacjami są:

* **INSERT** - służy do umieszczania danych w tabeli,
* **UPDATE** - odpowiada za zmianę danych
* **DELETE** - usuwa dane

**DDL - Data Definition Language**

Umożliwia dodawanie, zmienianie i usuwanie tabel lub bazy, przykładami poleceń z powyższej grupy są:

* **CREATE** - tworzenie struktury (np. bazy, tabeli)
* **DROP** - usunięcie struktury
* **ALTER** - zmiana struktury (np. dodanie kolumny do tabeli)

**DQL - Data Query Language**

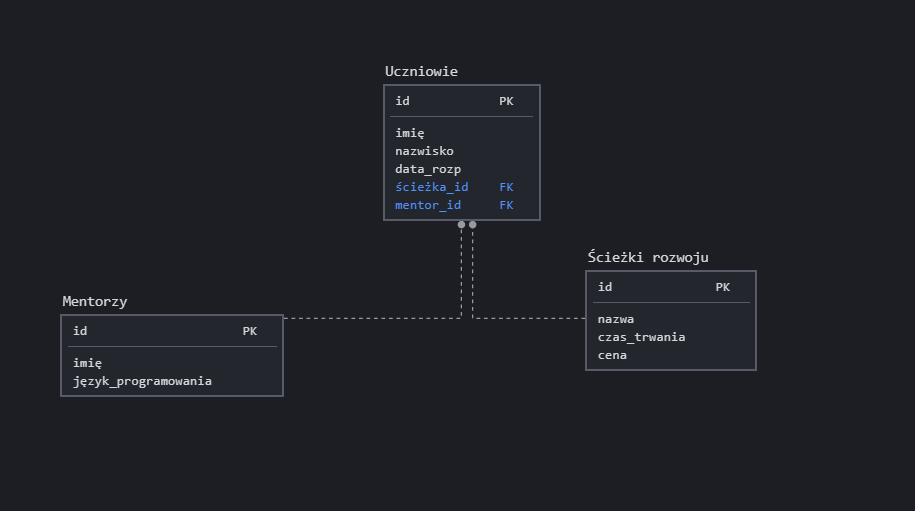
W jego zakres wchodzi polecenie - **SELECT**. Jego praktyczne wykorzystanie zobaczysz w sekcji tego szkolenia, gdzie odwzorowywać będziemy tabele Uczniowie, Mentorzy, Ścieżki rozwoju i relacje między nimi.

Zanim przejdziemy do następnej sekcji, chciałbym podsumować to wszystko, co poznaliśmy do tej pory. SQL to zbiór angielskich słów tworzących określone zapytania (funkcyjny język wysokopoziomowy). Tak więc wiele funkcjonalności można realizować ‘na wyczucie’, bez konieczności zapamiętywania dokładnie każdego polecenia z osobna.

**Zastosowanie DDL, DML, DQL w praktyce**

Załóżmy, że chcemy odzwierciedlić strukturę Naszych uczniów, ścieżek rozwoju, z jakich oni korzystają i do jakiego mentora są przydzieleni w ramach mentoringu Devs-Mentoring.pl.

Powyższą ideę możemy przedstawić na takim diagramie:



Na rzecz przykładu, przytoczymy na razie tylko i wyłącznie same polecenia SQL, bez integracji z Pythonem. Jeżeli chcesz je przetestować, to skorzystaj z silnika zapytań umieszczonego na stronie: [**https://sqliteonline.com/**](https://sqliteonline.com/)

Zakładając, że korzystamy z implementacji SQLite (programując już w Pythonie, będziemy korzystali właśnie z niej znajdującej się w bibliotece sqlite3), powyższe tabele z danymi możemy odzwierciedlić, wykorzystując kod tak jak poniżej:

**Tworzenie tabel i ustalanie powiązań między nimi**

Zauważ, że jest to grupa komend należących do grupy DDL.

**Przykład:** Utworzenie tabeli Mentorzy

**CREATE** **TABLE** **IF** **NOT** **EXISTS** Mentorzy (

id **integer** **PRIMARY** **KEY**,

imię **text** **NOT** **NULL**,

język\_programowania **text**

);

**Przykład:** Utworzenie tabeli Ścieżki rozwoju.

**CREATE** **TABLE** **IF** **NOT** **EXISTS** 'Ścieżki rozwoju' (

id **integer** **PRIMARY KEY**,

nazwa **text** **NOT NULL**,

czas\_trwania **integer** **NOT NULL**,

cena **double**

);

**Przykład:** Utworzenie tabeli Uczniowie.

**CREATE** **TABLE** **IF** **NOT** **EXISTS** Uczniowie (

id **integer** **PRIMARY KEY**,

imię **text** **NOT NULL**,

nazwisko **text** **NOT NULL**,

data\_rozp **DATE NOT NULL**,

ścieżka\_id integer,

mentor\_id integer,

**FOREIGN KEY** (ścieżka\_id) **REFERENCES** 'Ścieżki rozwoju'(id),

**FOREIGN KEY** (mentor\_id) **REFERENCES** Mentorzy(id));

**UWAGA:**

1. SQL nie jest językiem case-sensitive, czyli jego polecenia możesz pisać zarówno małymi jak i wielkimi literami i nie wpłynie to na działanie programu.
2. Jeżeli chcesz wykorzystać nazwę danej tabeli, która ma w sobie spację, musisz zawrzeć ją w apostrofach/cudzysłowach, np. zamiast Ścieżki rozwoju konieczny jest zapis 'Ścieżki rozwoju'.

**Tworzenie powiązań**

Pierwsza rzecz, na którą należy zwrócić uwagę, jest to w jaki sposób utworzyliśmy **powiązania między tabelami** (przypominasz sobie pojęcie Foreign Key, relacja jeden-do-jeden itd?).

Z diagramu możesz odczytać, że mamy do czynienia z powiązaniem **jeden do wielu** między tabelami: Ścieżki rozwoju - Uczniowie oraz Mentorzy - Uczniowie (np. do jednego ucznia możemy przypisać tylko jedną ścieżkę, ale do jednej ścieżki kilkoro uczniów).

Tworzenie takiego powiązania jest możliwe, umieszczając w danej tabeli **klucz obcy (Foreign Key)**, np. ścieżka\_id w tabeli Uczniowie i na końcu umieszczając polecenie:

**FOREIGN KEY** (ścieżka\_id) **REFERENCES** 'Ścieżki rozwoju'(id).

Innymi słowy, ścieżka\_id odnosi się do **id ścieżki rozwoju** i w ten sposób kojarzona jest dowolna krotka, reprezentująca ucznia, ze ścieżką, jaką on podąża.

Zwróć również uwagę na to, gdzie znajdują się przecinki, nawiasy oraz średniki w przytoczonym przykładzie.

Tak jak w przypadku Pythona - najprostsze błędy składniowe mogą mocno utrudniać życie i powodować 'zonk' dla interpretera poleceń SQL.

**Właściwości danych**

Spójrz, że przy dodawaniu atrybutów do tabeli (np. id, imie, nazwisko itd.) podaliśmy pewne **właściwości tych danych** (np. PRIMARY KEY, NOT NULL) Określają one cechy danych pól, np. **PRIMARY KEY** oznacza, że dany atrybut ma być traktowany jako **klucz główny** (jeżeli nie pamiętasz, do czego odnosi się to pojęcie, zerknij na sekcje wyżej), a **NOT NULL** dotyczy tego, że wartość pola nie **może być pusta** i przy tworzeniu obiektu w danej tabeli, musimy określać jego wartość.

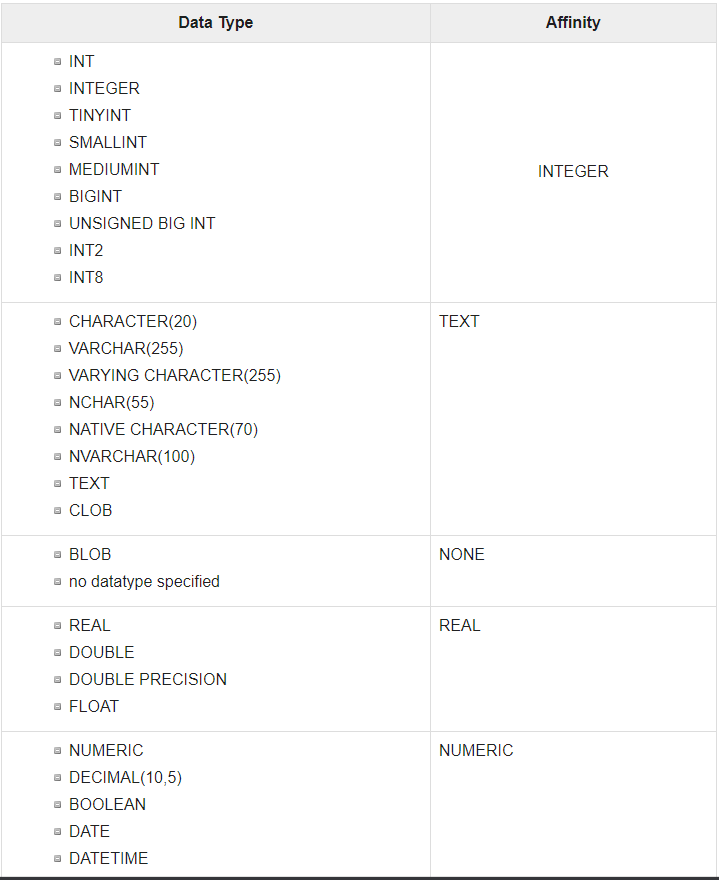
Jeżeli chodzi o SQLite, mamy również do dyspozycji takie właściwości pól jak:

* **PrimaryKey -** poznałeś już tę własność. Ustawia ona dany atrybut jako klucz główny (czyli wartość unikalną i identyfikującą każdy obiekt danych, np. if ucznia, nr identyfikacyjny szkolenia itd).
* **AutoIncrement** - właściwość, która często jest wykorzystywana w połączeniu PrimaryKey. Ma na wpły taki, że przy wstawianiu kolejnych danych do tabeli (polecenie INSERT), **automatycznie zwiększa się** wartość danego pola liczbowego o 1. Moglibyśmy zatem przypisać tę właściwość do ID ucznia i przez to zostalibyśmy zwolnieni z ręcznego ustawiania coraz to większych wartości ID dla kolejno dodawanych obiektów w tabeli.
* **Indexed -** służy do indeksowania danego pola. Przypisanie tej właściwości do danego atrybutu, przyspiesza proces wyszukiwania jego wartości (szczególnie przydatne, gdy mamy mega obszerne tabele i zależy Nam na efektywności operacji).

**Typy Danych**

Zauważyłeś zapewne również, że oprócz właściwości przy danym polu, określamy również z jakim typem danej mamy do czynienia (np. integer, text, double).

Poniżej ściąga zawierająca pozostałe typy, jakie można wykorzystywać w SQLite:



Zauważ, że lewa kolumna to tak naprawdę podkategoria prawej kolumny, np. TINYINT i SMALLINT należą do typu INTEGER. Oczywiście, przy określaniu typów możesz ograniczać się tylko do **kolumny Affinity**, ale chcąc jeszcze dokładniej określać charakter danego typu (np. ile pamięci ma zająć w bazie, jakiego ma być zakresu), to wówczas wykorzystasz typy z lewej kolumny.

**Wypełnianie tabeli danymi**

W porządku - mamy utworzone tabele (Mentorzy, Ścieżki rozwoju, Uczniowie), ale póki co są one puste - tzn. nie przechowują żadnych danych.

Oczywiste jest, że wraz z rozwojem kursu, chcielibyśmy dodawać coraz to nowych uczniów mentorów i ścieżki rozwoju. Aby to zrealizować, tym razem skorzystamy z **grupy poleceń DML** (a dokładniej **INSERT**)!

**Dodawanie mentorów do tabeli:**

1. **INSERT INTO** Mentorzy(id, imię, język\_programowania) **VALUES**(1, 'Jan', 'Python i C++');
2. **INSERT INTO** Mentorzy(id, imię) VALUES(2, 'Kacper');

W wyniku powyższych operacji, do tabeli Mentorzy zostaną umieszczeni mentorzy o imionach **Jan** i **Kacper**. Zwróć uwagę na niewielką różnicę między tymi poleceniami.

Drugie jest krótsze, ponieważ nie określamy w nim, w jakim języku programowania specjalizuje się mentor. Oczywiście zmiana tego pola jest możliwa w dowolnym momencie (przy wykorzystaniu polecenia **UPDATE**, o którym za chwilę), jednak różnice w składni poleceń pokazują, że nie zawsze musimy wypełniać **wszystkich atrybutów** danego obiektu w tabeli.

Jedynym jednak warunkiem, aby móc pominąć inicjalizacją danego pola jest nieumieszczenie właściwości NOT NULL przy danym polu (spójrz na etap tworzenia tabeli przy użyciu polecenia CREATE).

**Dodawanie ścieżek rozwoju do tabeli:**

1. **INSERT INTO** 'Ścieżki rozwoju'(id, nazwa, czas\_trwania, cena) **VALUES**(1, 'Podstawowa', 4, 560);
2. **INSERT INTO** 'Ścieżki rozwoju'(id, nazwa, czas\_trwania, cena) **VALUES**(2, 'Średniozaawansowana', 8, 860);
3. **INSERT INTO** 'Ścieżki rozwoju'(id, nazwa, czas\_trwania, cena) **VALUES**(3, 'Zaawansowana', 12, 1000);

**Dodawanie uczniów do tabeli:**

1. **INSERT INTO** Uczniowie(id, imię, nazwisko, data\_rozp, ścieżka\_id, mentor\_id) **VALUES**(1, 'Adrian', 'Chleba', '2020-05-01', 1, 2);
2. **INSERT INTO** Uczniowie(id, imię, nazwisko, data\_rozp, ścieżka\_id, mentor\_id) **VALUES**(2, 'Karol', 'Janeczek', '2020-06-01', 2, 1);
3. \*\***INSERT INTO** Uczniowie(id, imię, nazwisko, data\_rozp, ścieżka\_id, mentor\_id) **VALUES**(3, ‘Jan', 'Lelek', '2020-07-01', 5, 1);

To, na co należy zwrócić uwagę w powyższych poleceniach, to w jaki sposób **podajemy datę**. Musisz podawać ją w formacie **YYYY-MM-DD** (ponieważ atrybut jest typu DATE).

Zastanawiasz się zapewne, dlaczego przykład 3 został oznaczony czerwonymi gwiazdkami.

To dlatego, że polecenie 3 spowoduje **błąd w dodawaniu rekordu** (nieprawidłowość klucza obcego dla tabeli 'Ścieżki rozwoju').

Zauważ bowiem, że próbujemy ucznia o **ID 3** powiązać z ścieżką **o ID 5**. Błąd wynika z tego, iż **jedyne ścieżki**, które dodaliśmy do tabeli Ścieżki Rozwoju **mają ID 1, 2, 3**.

Wpisując zatem wartość 5 jako id\_ścieżki w tabeli Uczniowie, próbujemy odwołać się do nieistniejącej instancji danych.

**UWAGA:**

Klucz PK (Primary Key) w danej tabeli jest unikalny. To znaczy, że nie możemy do danej tabeli dodać dwóch rekordów o takim samym kluczu głównym.

Zatem błędem byłoby zapisanie czwartego polecenia w postaci:

1. **INSERT INTO** Uczniowie(id, imię, nazwisko, data\_rozp, ścieżka\_id, mentor\_id) **VALUES**(3, ‘Grzegorz', ‘Gżegżółka', '2018-07-01', 5, 1);

Zauważ, że w takim błędnym przypadku - Grzegorz Gżegżółka miałby ten sam ID (równy 3), co Karol Janeczek.

**Pozostałe przypadki poleceń**

Super, udało Nam się stworzyć tabele i umieścić w nich dane. W zasadzie możemy powiedzieć, że mamy już w pełni funkcjonalną bazę danych. Nic tylko stawiać ją na jakimś serwerze i integrować API do jej obsługi : )

Wracając jednak na ziemię, przyznaj, że z czasem może nastąpić potrzeba choćby modyfikowania atrybutów tabeli, usuwania tabel i użytkowników, zmieniania informacji o dodanych użytkownikach, wyświetlania informacji o dodanych użytkownikach etc.

Dlatego konieczne jest poznanie pozostałych, równie ważnych, co wcześniej zaprezentowane, poleceń z DML, DDL i DQL. Prześledźmy ich działanie na przykładach.

**DML - pozostałe przykłady:**

**UPDATE**

UPDATE służy do aktualizacji rekordów. UPDATE moglibyśmy wykorzystać, chcąc, np. zmienić cechy wybranych krotek. Co ciekawe, UPDATE niekoniecznie musi służyć do modyfikowania danych jednego obiektu (np. jednego ucznia), ale może również zmieniać właściwości kilku grup obiektów (np. zmiana roku urodzenia wszystkich użytkowników, których PESEL rozpoczyna się od cyfr 00).

**Zmiana imienia ucznia o określonym ID:**

**UPDATE** Uczniowie **SET** imię = "Karolek" **WHERE** id = 2;

**Zmiana kilku danych ucznia o określonym ID:**

**UPDATE** Uczniowie **SET** imię = "Janek", data\_rozp = '2021-01-01' **WHERE** id = 1;

**Zmiana wszystkich rzędów danych:**

**UPDATE** Pracownicy **SET** email **= LOWER(** imie || "." || nazwisko || "@chinookcorp.com")**;**

**UWAGA:**

1. **imie**, **nazwisko** to atrybuty tabeli Pracownicy, a **||** to operatory łączenia wyrazów; powyższe polecenie spowoduje więc utworzenie emaili zgodnie z wzorem: *imie.nazwisko@chinookcorp.com*.
2. Zauważ również, że w poleceniu UPDATE wykorzystujemy operator WHERE. Więcej na ten temat znajdziesz w dalszej części kursu dotyczącej polecenia SELECT z rodziny DQL.

**DELETE**

Nadszedł czas na usuwanie krotek (np. ucznia, który zrealizował już swój cel). Podobnie jak w przypadku UPDATE, usuwać możemy zarówno pojedynczy rekord jak i całą grupę spełniającą dany warunek.

**Usuwanie pojedynczego rekordu**

**DELETE FROM** Uczniowie **WHERE** id = 1;

**Usuwanie rekordów uczniów, których nazwisko zaczyna się na P:**

**DELETE FROM** Uczniowie **WHERE** nazwisko **LIKE** 'P%';

**Usuwanie wszystkich rekordów tabeli (tabela pozostanie pusta):**

**DELETE FROM** Uczniowie;

**Operator LIKE**

Z tego miejsca warto poruszyć temat użytego w przykładzie (usuwanie rekordów uczniów, których nazwisko zaczyna się na P) operatora wykorzystywanego w połączeniu z tekstem (napisem).

Nie zawsze bowiem chcemy operować na danych, które mają konkretną, z góry ustaloną zawartość. Naszym celem może być bowiem zmodyfikowanie, np. wszystkich rekordów, **które zaczynają się na określoną literę**, mają **w środku dany ciąg znaków** itd.

Aby umożliwić przetwarzanie danych, które spełniają dany wzorzec można skorzystać właśnie z operatora **LIKE**.

Co istotne, przy określaniu wzoru danego tekstu, w jego treści umieszczamy **%** lub **\_**.

Są to tzw. **"SQL wildcards"**:

* Znak % dopasowuje **dowolną ilość liter w danym ciągu** (0 lub więcej)
* Znak \_ odpowiada **jednej pojedynczej literze**

Różnicę między powyższymi wildcard'ami lepiej zrozumiesz na pewno na przykładach:

* s% - odpowiada dowolnemu stringowi, który zaczyna się na literę **s** i ma **dalej dowolne litery** (lub ich brak)
* s\_ - odpowiada dowolnemu dwuelementowemu napisowi, który zaczyna się na s
* %per% - odpowiada (wybacz za angielskie przykłady) słowom pep**per, per**cent
* natomiast h\_nt słowom **h**u**nt**, **h**i**nt**

**DDL - pozostałe polecenia:**

**- DROP - usuwanie tabeli**

Wyobraźmy sobie, że na pewnym etapie Naszej pracy, okazało się, że jednak nie potrzebujemy tabeli Mentorzy.

Aby ją usunąć, wykonamy polecenie DROP:

**DROP TABLE IF EXISTS** Mentorzy;

**- ALTER - modyfikowanie struktury tabeli**

ALTER (ang. zmienić) służy głównie do modyfikowania informacji w bazie. Co istotne jednak, ALTER **nie służy do modyfikowania danych danego obiektu** (np. zmiana imienia Kacper na Janek, nazwiska na Kowalski itd), **ale nazw kolumn** (atrybutów) lub **nazw tabeli**.

**Zmiana nazwy tabeli**

Nawiązując do wcześniejszego przykładu, aby nie podejmować drastycznych kroków usuwania tabeli, możemy po prostu zmienić jej nazwę tak jak poniżej:

**ALTER** **TABLE** Mentorzy **RENAME** **TO** Mentorzy\_Obsolete;

**Zmiana nazwy kolumny**

**ALTER TABLE** nazwa\_tabeli **RENAME COLUMN** stara\_nazwa **TO** nowa\_nazwa;

**Dodawanie kolumny**

Chcąc zmodyfikować daną tabelę i dodać do niej nową kolumnę, wykorzystamy:

**ALTER TABLE** nazwa\_tabeli **ADD COLUMN** nazwa\_kolumny;

**Usuwanie kolumny**

Okazuje się, że usuwanie kolumny w SQLite do wersji 3.35.0 języka nie było wcale tak trywialną operacją. Do tego czasu nie było bowiem możliwości wykonania polecenia DROP na kolumnie. Niedawno jednak się to zmieniło i od teraz, aby usunąć określoną kolumnę z tabeli, powinniśmy wykonać poniższe polecenie:

**ALTER TABLE** nazwa\_tabeli **DROP COLUMN** nazwa\_kolumny;

**DQL - przykłady:**

W końcu udało się dojść do chyba jednej z najczęściej wykorzystywanej grupy poleceń w SQL. Składa się ona tylko z jednego polecenia, którym jest SELECT. Służy ono bowiem do wyświetlania zawartości danych tabel jak i rekordów spełniających pewne warunki (np. wyświetlanie danych tylko i wyłącznie z grupy mężczyzn).

Wraz z poleceniem SELECT i wyświetlaniem danych, możemy również efektywnie wykorzystywać sporo nowych operatorów (takich jak ORDER BY, GROUP BY, LIMIT) o których szerzej za chwilę...

**Pobieranie wszystkich informacji z tabeli**

Pamiętasz jeszcze strukturę utworzonej przez Nas bazy danych składającej się z takich tabel jak: Uczniowie, Mentorzy, Ścieżki rozwoju? Załóżmy, że chcielibyśmy wyświetlić **wszystkich** zapisanych uczniów.

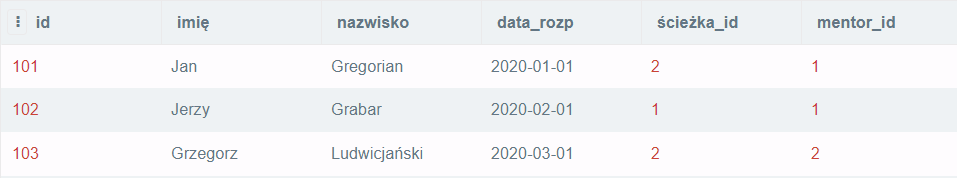
W tym celu wykorzystamy poniższą składnię zapytania SELECT:

**SELECT** \* **FROM** Uczniowie;

To co wymaga wyjaśnienia z powyższego polecenie to w zasadzie tylko **\*.** Zapamiętaj proszę, że gwiazdkę tę można tłumaczyć jako **(wszystkie/wszystko)**. Tak więc przekładając powyższe polecenie na bardziej zrozumiały język - jej działanie to:

*Wybierz (i wyświetl) wszystkie rekordy z tabeli Uczniowie*

**Wynik polecenia SELECT \* FROM Uczniowie;**



**Wyświetlanie poszczególnych atrybutów rekordu:**

Aby wyłuskać tylko niektóre informacje z zapisanych danych w tabeli, np. wyświetlić tylko imię i nazwisko ucznia, użyjemy składni:

**SELECT** nazwa\_kolumny1**,** nazwa\_kolumny2 **FROM Uczniowie;**

Czyli:

**SELECT** imię**,** nazwisko **FROM** Uczniowie**;**

**SELECT z użyciem operatorów**

Teraz nadeszła pora na sprecyzowanie warunków, które muszą spełnić rekordy danych (aby zostać, np. wyświetlonymi).

Posłużymy się operatorami:

1. **WHERE** - to operator, który jest tak naprawdę nieodłączny przy określaniu warunków, które muszą spełniać dane. Nie wiem, czy zwróciłeś uwagę, ale WHERE już wykorzystywaliśmy wraz z poleceniem **UPDATE**.   
   Wówczas używaliśmy go w momencie, gdy chcieliśmy wyświetlić ucznia o określonym id (**UPDATE** Uczniowie **SET** imię = "Karolek" **WHERE** id = 2;)WHERE przy SELECT ma w zasadzie takie samo zastosowanie, co w przypadku UPDATE.  
   Czyli chcąc, np. **wyświetlić dane użytkowników**, których imię zaczyna się na **literę K**, zapiszemy: **SELECT** \* **FROM** Uczniowie **WHERE** imię **LIKE** 'K%';
2. **AND & OR** - nie bez przyczyny umieszczamy ten operator zaraz po **WHERE**. Umożliwia on rozbudowywanie warunków dotyczących selekcji danych.

Prosty przykład - wyobraź sobie tabelę Klientów. Posiada ona m.in atrybuty takie jak: płeć, wiek.

Spośród wszystkich klientów, chcemy wyświetlić tylko kobiety, które są w wieku **między 30 a 40 lat**. W tym celu zapiszemy poniższe polecenie:

**SELECT** \* **FROM** Klienci **WHERE** płeć = 'K' **AND** wiek >= 30 **AND** wiek <= 40;

Chcąc natomiast wyświetlić tylko tych klientów którzy mieszkają we Wrocławiu lub Warszawie, użyjemy:

**SELECT** \* **FROM** Klienci **WHERE** miasto **LIKE** 'Wrocław' **OR** miasto **LIKE** 'Warszawa'

1. **ORDER BY** - umożliwia sortowanie wyników wg danego atrybutu, np. sortowanie danych wynikowych według nazwiska.   
   Jak się zapewne domyślasz, sortować możemy w sposób **niemalejący** jak i **nierosnący**. Dlatego w celu określenia, w jaki sposób dane mają być ułożone, oprócz samego **ORDER BY** użyjemy **DESC** (descending)oraz **ASC (**ascending - wartość domyślna).

Załóżmy, że chcemy posortować **rosnąco** według daty rozpoczęcia nauki wszystkich pobranych użytkowników, **których nazwisko ma więcej niż 10 liter**:

**SELECT** \* **FROM** Uczniowie **WHERE** length(nazwisko) > 10 **ORDER BY** data\_rozp **ASC;**

Chcąc przeprowadzić podobną operację, ale tym razem sortując tych samych uczniów **malejąco**:

**SELECT** \* **FROM** Uczniowie **WHERE** length(nazwisko) > 10 **ORDER BY** data\_rozp **DESC;**

1. **LIMIT** - używany, gdy chcemy ograniczyć ilość wyświetlanych rzędów danych, np. gdy chcemy pobrać **10 ostatnio** zapisanych użytkowników na kurs programowania.  
     
   **SELECT** \* **FROM**  Uczniowie **ORDER BY** data\_rozp DESC **LIMIT** 10;

**Uzupełnienie wiedzy**

Będąc uzbrojonym już w tak solidne podstawy i wiedzę domenową z zakresu SQL-a, czuje się zobowiązany, aby wspomnieć Ci o sposobach modelowania zależności między tabelami. Otóż bądź świadomy, że dowiezienie każdego projektu informatycznego wymaga porządnej inwestygacji oraz przeprowadzenia dogłębnej analizy wszystkich wymagań. Jednym słowem, zespół developerów odpowiedzialny za rozwój projektu, musi mieć solidne fundamenty oraz założenia, na których będzie bazował podczas implementacji.

Dlatego też konieczne jest stworzenie m.in schematu bazy (czy też ogólnie - klas aplikacji), który odzwierciedlać będzie strukturę bazy. Umożliwi to obranie wspólnych celów i łatwiejszą implementację założeń. Do tworzenia takich schematów często wykorzystuje się diagramy UML.

Diagramy te to po prostu graficzne odzwierciedlenie, np. tabel/klas, które będą implementowane w programie. Istnieją jednak pewne reguły, których należy się trzymać, tworząc diagramy. Zasady te gromadzone i określane są właśnie przez UML (Unified Modeling Language). Zalecamy zatem zapoznanie się ze wspomnianym tematem i wykorzystanie takich źródeł wiedzy jak:

* [Podstawy UML](https://www.samouczekprogramisty.pl/podstawy-uml/)
* [Welcome To UML Web Site!](https://www.uml.org/)
* [Unified Modeling Language – Wikipedia, wolna encyklopedia](https://pl.wikipedia.org/wiki/Unified_Modeling_Language)

**Zakończenie**

Tymi o to kilkunastoma treściwymi stronami, kończę dzisiejsze szkolenie. Miało ono na celu pokazanie Ci podstawowych poleceń z zakresu SQLite’a oraz wdrożenie Cię w tematykę baz danych.

Jako ćwiczenie, Twoim zadaniem jest na pewno realizacja wszystkich podpunktów tego kursu. Śmiało testuj różne możliwości, przyzwyczajaj się do błędów wypluwanych przez interpreter poleceń. Tak, jak już wspomniałem do testowania funkcjonalności bazy SQLite, polecam Ci narzędzie [**https://sqliteonline.com/**](https://sqliteonline.com/)

Zapoznaj się dokładnie z możliwościami poleceń, tak aby móc efektywnie implementować je w Pythonie. Przed Nami przecież omówienie biblioteki sqlite3 wykorzystywanej przez ten język.