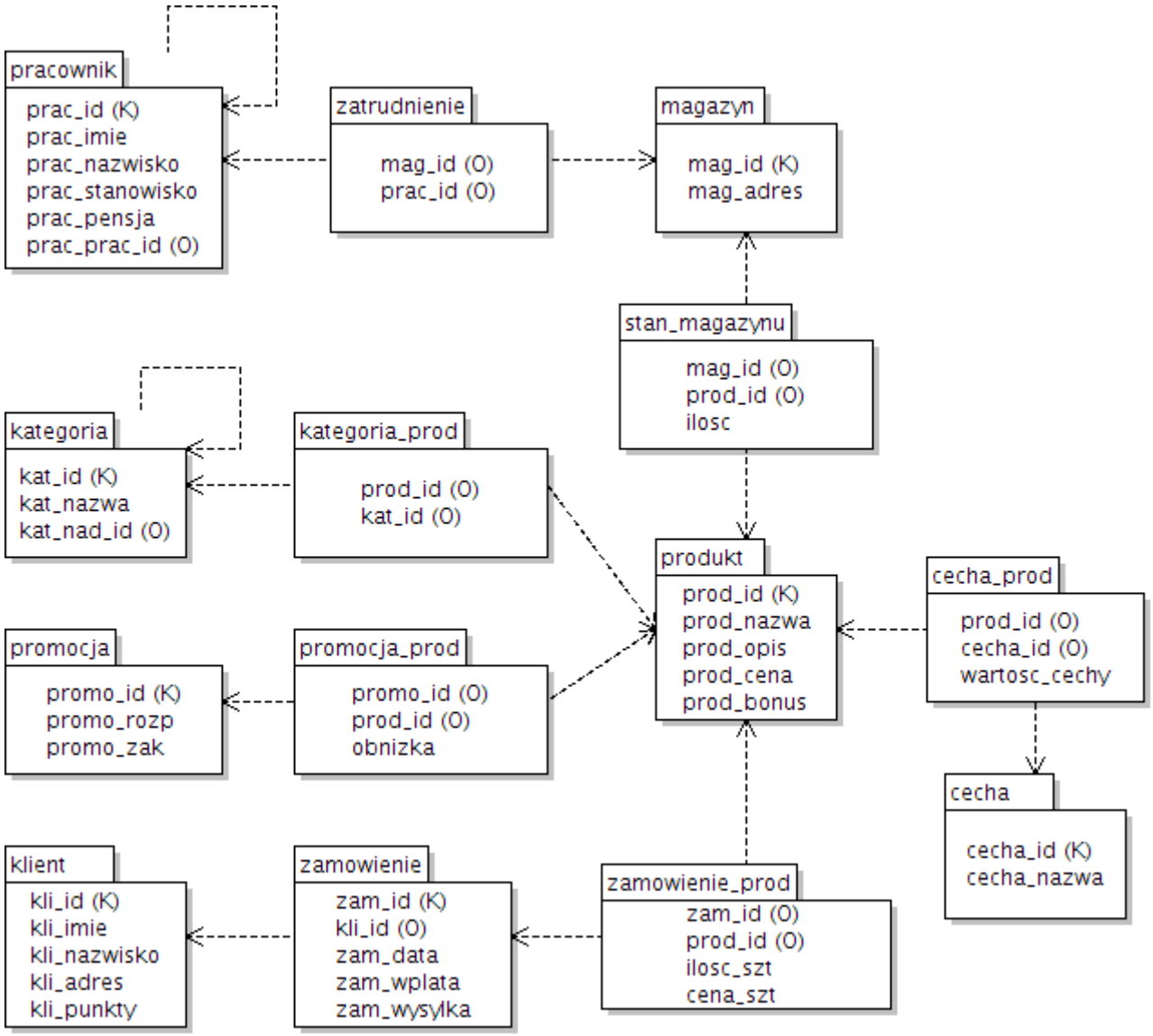
Maciej Stefańczyk, gr. J1I2 *16 grudnia 2008r.*

BAZY DANYCH

OPRACOWANIE – SPRAWDZIAN 1

Na ćwiczeniach laboratoryjnych zadaniem było stworzenie bazy danych do obsługi sklepu (lista pracowników, klientów, produktów, magazyny z produktami, promocje). Poniżej przedstawiam tabele występujące w mojej bazie (literą K oznaczam klucz w danej tabeli, literą O oznaczam klucze obce, strzałkami zależności pomiędzy poszczególnymi tabelami).



Zad. 1

Normalizacja bazy danych

Na początek przeprowadzę analizę znormalizowania relacji w mojej bazie danych, uwzględniając pierwsze trzy postaci normalne.

Tabele w relacyjnej bazie danych automatycznie spełniają większość warunków 1NF (jeśli chodzi o te warunki, to jest ich dużo i nie ma jednej ustalonej definicji pierwszej postaci normalnej...). Warunkiem, którego nie spełnia część relacji w mojej bazie jest brak wartości mogących przyjmować wartość NULL – w związku z czym tabele *pracownik*, *kategoria* oraz *zamowienie* nie spełniają warunków 1NF.

Warunki drugiej postaci normalnej wymagają, aby każde pole w tabeli nie należące do klucza było zależne od całego klucza (jeśli klucz składa się z kilku kolumn), a nie od jego części. Oprócz tego oczywiście tabela musi być też w 1NF. Jeśli tabela ma klucz składający się z jednego pola, to automatycznie jest w 2NF (*magazyn, produkt, promocja, cecha, zatrudnienie, kategoria_prod*). Pozostałe tabele (*stan_magazynu, cecha_prod, zamowienie_prod, promocja_prod*) mają klucz składający się z dwóch kolumn. Np. w tabeli przechowującej spis produktów z zamówień jednoznacznym identyfikatorem każdego rekordu jest para wartości {*zam_id*, *prod_id*}. Ilość sztuk danego produktu zależy od obydwu tych pól (w innym zamówieniu jeśli wystąpi ten sam produkt może mieć inną ilość, jeśli w tym samym zamówieniu wystąpi inny produkt również może mieć inną ilość). Podobnie cena jednostkowa produktu w momencie zakupu – zależy od id zamówienia (a konkretniej od daty złożenia tego zamówienia) oraz od zamawianego produktu.

Relacją, które nie spełnia założeń 2NF mogłaby być tabela przedstawiona w następnym punkcie (spis towarów w magazynach przechowujący jednocześnie adresy magazynów) Kluczem w niej jest para wartości {*mag_id, prod_id*}, jednak istnieje też pole *mag_adres*, które zależy tylko i wyłącznie od części tego klucza (id magazynu), a nie ma na nie wpływu druga część klucza (czyli id produktu). Taka zależność funkcyjna, a więc sytuacja, w której jakieś pole nie należące do klucza zależy jedynie od części tego klucza a nie od całości, może być nazwana „niepożądaną" i świadczyć o możliwości wystąpienia różnych problemów i niespójności podczas użytkowania bazy.

Warunkiem, który relacja musi spełnić, aby należeć do 3NF (oczywiście oprócz bycia w 2NF), jest zależność atrybutów tylko i wyłącznie oraz bezpośrednio od całego klucza. Jeśli istnieje w tabeli pole jedynie pośrednio zależne od klucza – relacja nie jest w 3NF. Wszystkie tabele, które u mnie spełniają 2NF są też w 3NF. Przykładem tabeli, która nie spełniałaby warunków może być np. tabela przechowująca zamówienia wraz z nazwiskiem klienta (*zam_id, kli_id, kli_nazwisko, zam_data, ...*). Tabela ta jest w 2NF (ponieważ jej klucz składa się tylko z pola *zam_id*), jednak pole przechowujące nazwisko klienta jest jedynie pośrednio zależne od klucza: *zam_id* → *kli_id* → *kli_nazwisko*. Taka zależność również może mieć niepożądane skutki w przyszłości (np. niespójność danych).

Jak widać, dopiero spełnienie przynajmniej tych trzech grup warunków gwarantuje w dobrym stopniu uwolnienie się od problemów związanych z anomaliami i duplikacją danych w tabelach, jednak nie zawsze tak się dzieje (min. z tego powodu istnieje jeszcze klika kolejnych postaci normalnych usuwających kolejne pojawiające się problemy). Jednak w mojej bazie spełnienie 3NF przez większość tabel gwarantuje pozbycie się problemów.

Opis anomalii użytkowych występujących w tabeli nieznormalizowanej oraz ocena ich uciążliwości w przypadku bazy danych sklepu.

W przypadku, gdy w tabeli próbujemy umieścić zbyt wiele danych, możemy mieć do czynienia w kilkoma rodzajami anomalii związanych z ich późniejszym użytkowaniem. Dla zobrazowania opisywanego problemu można wyobrazić sobie tabelę przechowującą jednocześnie informacje o samym magazynie (takie jak jego adres) oraz informacje o towarach w danym magazynie. Taka relacja mogłaby mieć postać (wraz z przykładowymi danymi)

mag_id	**mag_adres**	**prod_id**	**ilosc**
1	Warszawa	1	4
1	Warszawa	2	5
2	Kraków	1	1
3	Wrocław	3	12

Na jej przykładzie można pokazać kilka głównych anomalii, które mogą się pojawić. Ocenę uciążliwości można przeprowadzić z punktu widzenia właściciela sklepu oraz z perspektywy klienta tego sklepu.

1. Anomalia modyfikacji

Jeśli adres jednego z magazynów ulegnie zmianie, konieczne będzie sprawdzenie każdego wpisu w tabeli i zmiana każdego wystąpienia tego adresu. Jeśli z powodu jakiegoś niedopatrzenia niektóre adresy zostaną zmienione, a niektóre pozostaną bez zmian, dane w tabeli utracą spójność (załóżmy, że zmiany dokonano tak, jak pokazuje poniższa tabela). Wg pierwszego rekordu magazyn o id = 1 znajduje się w Warszawie, drugi wpis zawiera jednak informację, że ten sam magazyn znajduje się w Pruszkowie.

Dla właściciela sklepu sytuacja taka jest niedopuszczalna i należy jej za wszelką cenę zapobiegać.

Dla klienta ta anomalia (w prezentowanym przypadku) jest w zasadzie niezauważalna. Gdyby jednak rozważyć nieznormalizowaną tabelę zawierającą dane klienta oraz np. jego zamówienia, po takiej nieudanej modyfikacji adresu klienta niektóre zamówienia by do niego nie dotarły.

mag_id	**mag_adres**	**prod_id**	**ilosc**
1	Warszawa	1	4
1	Pruszków	2	5
2	Kraków	1	1
3	Wrocław	3	12

2. Anomalia wstawiania

Jeśli nasza firma zakupi nowy magazyn i będzie chciała wstawić do bazy informacje o nim, zatrzymamy się w momencie decyzji co wpisać w pola prod_id oraz ilosc. Jako że nasz magazyn jest dopiero co zakupiony, to nie ma w nim żadnych produktów. Można oczywiście wstawić dowolny numer produktu i jego ilość ustawić na zero, jednak takie rozwiązanie jest tylko półśrodkiem i obejściem problemu, w większości sytuacji nie da się go zastosować.

Dla właściciela sklepu główną uciążliwością związaną z tą anomalią jest konieczność dodatkowej pracy podczas dodawania nowego magazynu.

Jeśli rozważymy opisaną wcześniej tabelę zawierającą dane klienta oraz jego zamówienia, anomalia ta pozbawia klienta możliwości zarejestrowania się w sklepie bez składania zamówień.

mag_id	**mag_adres**	**prod_id**	**ilosc**
1	Warszawa	1	4
1	Warszawa	2	5
2	Kraków	1	1
3	Wrocław	3	12
+			
4	Lublin	???	???

3. Anomalia usuwania

Jeśli z magazynu nr 2 usuniemy wszystkie produkty, z bazy znikną wszystkie informacje o tym magazynie (usuwany wiersz tabeli był jedynym miejscem przechowywania informacji o tym magazynie). Po tej operacji niemożliwe stanie się dodanie produktów do tego magazynu (jako, że nie będzie już o nim informacji w bazie). Tutaj ponownie rozwiązaniem może być zerowanie pola ilości towaru, jednak – tak jak już napisałem w poprzednim punkcie – jest to jedynie proteza a nie prawidłowe rozwiązanie problemu.

Właściciel sklepu, po stracie informacji o swoim magazynie, musi ją ręcznie wprowadzić podczas następnej operacji aktualizacji stanów magazynowych.

Rozważmy teraz tabelę historii zamówień w której przechowujemy dane klientów oraz złożone przez nich zamówienia. Jeśli klient złoży swoje pierwsze zamówienie – musi przejść przez pełną procedurę rejestracji. Jeśli jednak już po zatwierdzeniu zamówienia zrezygnuje z niego i będzie chciał złożyć inne, dopiero co wprowadzone do bazy informacje o kliencie znikną i będzie konieczne ponowne przejście całej procedury rejestracyjnej. Z czysto marketingowego punktu widzenia – takie zachowanie naszej bazy będzie zniechęcało klientów do korzystania z naszych usług, a więc będzie uciążliwe także dla właściciela.

mag_id	**mag_adres**	**prod_id**	**ilosc**
1	Warszawa	1	4
1	Warszawa	2	5
~~2~~	~~Kraków~~	~~1~~	~~1~~
3	Wrocław	3	12

Ocena kosztów rozwiązania znormalizowanego oraz kryteria decyzji o normalizacji.

Oprócz wymienionych wad, nieznormalizowane relacje mają również kilka zalet. Największą z nich jest szybkość dostępu do pełnych danych związanych np. z zamówieniem. Za pomocą prostego zapytania otrzymujemy jednocześnie dane klienta i zamówione przez niego produkty. W celu otrzymania tego samego wyniku z relacji znormalizowanych konieczne jest wykonywanie zapytań ze złączeniem tabel, a operacja ta zajmuje dodatkowy czas. W przypadku przedstawionych wyżej tabel różnica jest niezauważalna i całkowicie pomijalna, jednak jeśli porównamy szybkość odczytu tysiąca kolumn z pojedynczej tabeli z szybkością zapytania, w którym łączymy najpierw 300 innych tabel i dopiero wyciągamy potrzebne nam dane – różnica może być zauważalna. Jeśli bardzo zależy nam na czasie działania normalizacja jest wręcz niewskazana.

W przypadku omawianej bazy danych sklepu czas jest czynnikiem drugoplanowym. Po pierwsze nie ma wielu baz, a więc łączeń będzie stosunkowo niewiele, po drugie to, czy strona z np. informacjami o aktualnej ofercie sklepu wyświetli się w sekundę czy w 2 sekundy nie stanowi dla klienta dużej różnicy – często opóźnienia większe od czasu dostępu do bazy wprowadza samo łącze internetowe.

Dużo ważniejszym czynnikiem przemawiającym za potrzebą normalizacji jest ilość miejsca zajmowana przez nieznormalizowane relacje, czyli **nadmiarowość** danych. Jeśli w magazynie przechowywane jest 1000 rodzajów produktów, w każdym wierszu tabeli przechowywane będą także pełne dane o magazynie, które same w sobie zajmują dużo więcej miejsca niż informacja o ilości towarów (różnica wielkości pomiędzy typem całkowitoliczbowym a typem znakowym może być kilkudziesięciokrotna). W przypadku tabeli znormalizowanej informacje o magazynie są przechowywane tylko w jednym miejscu, więc oszczędzamy miejsce zajmowane niepotrzebnie przez 999 wpisów. Jeśli nie posiadamy własnego serwera baz danych, ilość zajmowanego miejsca jest krytyczna – opłata za serwer często jest zależna od wielkości naszej bazy.

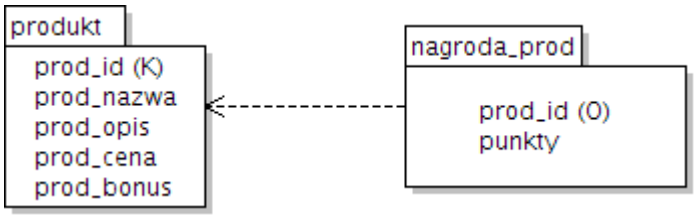
Biorąc pod uwagę powyższe uzasadnienie – w przypadku mojej bazy danych przeprowadzona normalizacja jest definitywna i ostateczna (w przypadku większości tabel).

Zad. 2

a) Modyfikacja struktury bazy danych dla uniknięcia konieczności stosowania znaczników NULL dla atrybutów opcjonalnych

W celu zilustrowania zagadnienia dodajmy do tabeli z produktami pole *wartosc_punktowa*, przechowujące informacje o tym, za ile punktów bonusowych można dostać w nagrodę dany przedmiot. Jeśli produktu nie można dostać za punkty, w polu wpisujemy wartość NULL.

Dla uniknięcia stosowania wartości NUL w tym polu można stworzyć dodatkową relację:



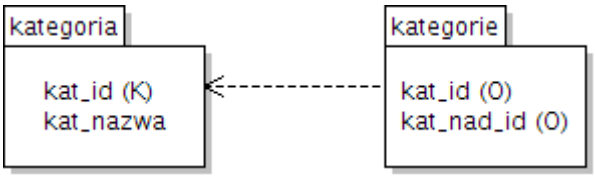
Jeśli jakiś produkt jest dostępny w programie partnerskim (można go „kupić" za punkty) dodajemy do tabeli nagroda_*prod* rekord zawierający id produktu oraz jego wartość punktową. Produkty, których nie można dostać za punkty w tej tabeli po prostu nie wystąpią. Bardzo ważnym warunkiem w tym przypadku jest unikalność pola *prod_id* w relacji *nagroda_prod*. W przeciwnym wypadku mogłoby dojść do niespójności danych – jeden wpis zawierałby informację, że produkt o id=1 można dostać za 100 punktów, w innym wierszu zawarta byłaby jednak informacja, że ten sam produkt można otrzymać za 200 punktów.

Pokazana modyfikacja (dodanie możliwości wyboru produktów z oferty sklepu za zebrane punkty, a także techniczna jej część – dodanie nowej relacji) być może zostałaby zrealizowana w późniejszym czasie, zależnie od popularności programu partnerskiego i decyzji właściciela sklepu.

b) Modyfikacja struktury bazy danych dla uniknięcia konieczności stosowania znaczników NULL w kluczu obcym w przypadku związku opcjonalnego.

W zaprojektowanej przeze mnie bazie danych większość występujących kluczy obcych musi być bezwarunkowo wypełniona, są tylko dwa klucze z wartością opcjonalną. Pierwszym z nich jest tabela opisująca pracownika i klucz obcy wskazujący na id bezpośredniego przełożonego (jeśli pracownik jest prezesem całej firmy, prac_prac_id = null), oraz opis kategorii i wskazanie na kategorię nadrzędną (dla głównych kategorii sklepu takich jak RTV, multimedia, książki kat_nad_id = null). Oba przypadki są do siebie podobne, więc opiszę tylko modyfikację tabeli kategoria.

Żeby uniknąć stosowania znacznika null w kluczu obcym kat_nad_id należałoby dodać dodatkową relację przechowującą pary wartości id kategorii i id kategorii nadrzędnej. Kategoria główna po prostu w tej relacji nie wystąpi, a więc nie będzie potrzeby wykorzystywania znacznika null.



Pokazana modyfikacja nie została jednak ostatecznie wprowadzona do struktury bazy, ponieważ znacznik null w kluczach obcych będzie występował znikomą ilość razy (raz dla pracowników, bo tylko jedna osoba jest szefem wszystkich szefów, oraz kilkukrotnie dla kategorii, ponieważ liczba głównych kategorii jest mocno ograniczona). Koszt wprowadzenia i użytkowania bazy z wprowadzoną zmianą byłby więc większy niż koszt użytkowania bazy w obecnej postaci.

c) Implementacja atrybutu wielowartościowego w przypadku modelu czysto relacyjnego, w którym wartości atrybutów muszą być skalarami.

Atrybutem wielowartościowym są cechy produktów – produkt może mieć teoretycznie nieograniczoną liczbę cech.

Rozważmy więc tabelę przechowującą dane o produktach. Na początku zaprojektowane zostały podstawowe pola tej relacji, takie jak id, nazwa, opis, cena oraz bonus punktowy za zakup. Konieczne okazało się także przechowywanie informacji dodatkowych o każdym z produktów, takich jak przekątna ekranu dla telewizorów bądź gatunek muzyczny dla płyt Audio-CD. Przy asortymencie o ograniczonej szerokości można by zastosować rozwiązanie przedstawione poniżej, a więc dodać dodatkowe pola do tabeli z produktami i wypełniać tylko te odnoszące się do danego produktu, a resztę pozostawiać pustą.

-- produkty

id	**nazwa**	**opis**	**cena**	**bonus**	**przekątna**	**gatunek**
1	SyncMaster 206bw	(opis monitora)	750	10	20,1	
2	U2 - „Zoorpa”	(recenzja płyty)	59	1		rock

Bardzo szybko wychodzą jednak wady takiego rozwiązania – jeśli stworzymy sobie bazę produktów, a później zechcemy rozszerzyć naszą ofertę np. o ubrania, konieczne stanie się dodanie w tabeli nowej kolumny z rozmiarem. Kolejną wadą jest ogromna liczba niewykorzystanych pól, które mimo wszystko zajmują miejsce w bazie danych.

W celu pozbycia się tych problemów tabela przechowująca informacje o produktach została ograniczona do 5 podstawowych pól, których wypełnienie jest obowiązkowe, wprowadzona została tabela przechowująca dostępne dla produktów cechy oraz relacja łącząca produkty z cechami (produkt może nie mieć żadnej cechy dodatkowej, wtedy nie występuje w tej relacji, może mieć też wiele cech, wtedy dla każdej cechy dodawany jest jeden wiersz w tej relacji). Przedstawiona wyżej tabela po rozbiciu wygląda następująco:

-- produkty

prod_id	prod_nazwa	prod_opis	prod_cena	prod_bonus
1	SyncMaster 206bw	(opis monitora)	750	10
2	U2 - „Zoorpa”	(recenzja płyty)	59	1

-- cechy

cecha_id	cecha_nazwa
1	gatunek
2	przekątna
3	producent

-- cechy_prod

prod_id	cecha_id	wartosc_cechy
1	2	20,1
2	1	rock
2	3	Island Music
1	3	Samsung

W przedstawionym rozwiązaniu, jeśli zechcemy dodać jakieś nowe cechy (np. producenta) wystarczy dodać jeden wiersz do tabeli z cechami i dopisać połączenia tej cechy z istniejącymi produktami w odpowiedniej tabeli. Struktury tabel pozostają niezmienione.

d) Zastosowanie relacji w roli słownika – jako implementacji dynamicznie definiowanej dziedziny wartości.

W bazie danych w obecnej postaci (przedstawionej na str. 1) o statusie zamówienia świadczy zawartość pól *zam_data* (data złożenia zamówienia), *zam_wplata* (data wpłaty należności przez klienta) oraz *zam_wysylka* (data wysyłki zamówienia). Jeśli określona jest tylko data złożenia zamówienia, statusem zamówienia jest „zatwierdzone, oczekiwanie na wpłatę”, po dokonaniu wpłaty status zmienia się na „przyjęte do realizacji”, a po wysyłce status zmienia się na „wysłane”. Zaletą przyjętego rozwiązania jest jego prostota – bez żadnych dodatkowych operacji złączenia, jednym zapytaniem możemy wyciągnąć z bazy informacje o statusie zamówienia. Wadą jest jednak konieczność stosowania znaczników NULL oraz trudność w przypadku konieczności zmiany funkcjonalności tabeli – np. dodania nowego statusu zamówienia (odrzucone, dostarczone itp.). Żeby pozbyć się drugiej niedogodności, należy dodać nowe pole przechowujące status zamówienia oraz dodatkową tabelę, w której będą określone możliwe statusy dla zamówienia:

Każde zamówienie musi mieć jakiś status – więc pole *stat_id* w tabeli *zamowienie* nie może mieć wartości NULL. Przy takiej organizacji tabel w bazie możliwe jest bardzo łatwe dodanie nowego statusu zamówienia (wystarczy dodać nowy wiersz do tabeli *status*).

Jeśli chcielibyśmy dodatkowo wyeliminować konieczność stosowania znaczników NULL w polach związanych z datą zamówienia, wpłaty i wysyłki, należałoby zastosować rozwiązanie przedstawione w poprzednim podpunkcie, czyli zrealizować zdarzenia związane z zamówieniem jako atrybut wielowartościowy i określić takie zdarzenia jak data zamówienia, data wpłaty, data wysyłki.

e) Złożony warunek poprawności

Poniższy warunek sprawdza zamówienia i nie dopuszcza do sytuacji, w której jedna osoba zamawia więcej niż 10 produktów tego samego typu, w celu zabezpieczenia się przed zamianą handlu detalicznego w handel hurtowy. Ze sprawdzenia wykluczone są produkty tańsze niż 10zł (dopuszczamy możliwość zakupu np. 20 płyt CD-R)

```
CREATE ASSERTION anty_hurt
CHECK (
        NOT EXISTS (
                SELECT      "zam_id", "prod_id", sum("ilosc_szt")
                FROM        "zamowienie_prod"
                WHERE       "cena_szt" > 10
                GROUP BY    "zam_id", "prod_id"
                HAVING      sum("ilosc_szt") > 10
        )
)
```

Zad. 3

Przykłady poleceń SELECT z zagnieżdżeniami

a) Zapytanie wybierające klientów, którzy złożyli więcej niż 10 zamówień:

Zapytanie jednopoziomowe

```
SELECT      kli_id, kli_imie, kli_nazwisko
FROM        klient JOIN zamowienie USING ("kli_id")
GROUP BY    kli_id, kli_imie, kli_nazwisko
HAVING      COUNT(*) > 10
```

Z zagnieżdżonym zapytaniem skorelowanym:

```
SELECT      kli_id, kli_imie, kli_nazwisko
FROM        klient k
WHERE       10 < (
        SELECT      COUNT(*)
        FROM        zamowienie z
        WHERE       k.kli_id = z.kli_id
)
```

b) Zapytanie wybierające pracowników, którzy zarabiają więcej od średniej wszystkich podwładnych tego samego szefa (czyli od wszystkich pracowników tego samego działu)

Z zagnieżdżonym zapytaniem nieskorelowanym:

```
SELECT      prac_id, prac_imie, prac_nazwisko
FROM        pracownik pr, (
        SELECT      pra_prac_id, AVG(prac_pensja) as srednia
        FROM        pracownik
        GROUP BY    prac_prac_id
) sr
WHERE       pr.prac_pensja > sr.srednia AND
            pr.prac_prac_id = sr.prac_prac_id
```

Z zagnieżdżonym zapytaniem skorelowanym:

```
SELECT      prac_id, prac_imie, prac_nazwisko
FROM        pracownik pr1
WHERE       pr1.prac_pensja > (
        SELECT      AVG(prac_pensja)
        FROM        pracownik pr2
        WHERE       pr1.prac_prac_id = pr2.prac_prac_id
)
```