Baza programisty

Pragmatic Bookshelf

Antywzorce języka SQL

Zobacz, jak tego nie robić!

- Jak nie projektować modelu logicznego i fizycznego bazy danych?
- Jak nie zadawać zapytań SQL?
- Jak nie wytwarzać aplikacji?

Helion

Bill Karwin

Tytuł oryginału: SQL Antipatterns: Avoiding the Pitfalls of Database Programming

Tłumaczenie: Mikołaj Szczepaniak

ISBN: 978-83-246-3482-8

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/antysq.zip

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/antysq
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 10.
Błędy zaokrągleń

10,0 razy 0,1 prawie nigdy nie jest równe 1,0.
Brian Kernighan

Nasz szef prosi nas o wygenerowanie raportu podsumowującego koszty pracy programistów według projektów (koszt ma być wyznaczany na podstawie czasu poświęconego na usuwanie poszczególnych błędów). Każdy programista reprezentowany w tabeli `Konta` otrzymuje inną stawkę godzinową, zatem musimy nie tylko rejestrować liczby godzin potrzebnych do usunięcia poszczególnych błędów (w tabeli `Bledy`), ale też mnożyć te wartości przez atrybut `stawka_godzinowa` zdefiniowany dla wyznaczonych programistów.

Plik Błędy-zaokrągleń/wprowadzenie/cost-per-bug.sql

```
SELECT b.id_bledu, b.godziny * k.stawka_godzinowa AS koszt_bledu
FROM Bledy AS b
  JOIN Konta AS k ON (b.przypisany_do = k.id_konta);
```

Obsługa tego zapytania wymaga utworzenia nowych kolumn w tabelach `Bledy` i `Konta`. Obie kolumny powinny obsługiwać wartości ułamkowe, ponieważ musimy śledzić koszty możliwie precyzyjnie. Decydujemy się więc zdefiniować nowe kolumny jako `FLOAT`, ponieważ właśnie ten typ danych obsługuje wartości ułamkowe.

Plik Błędy-zaokrągleń/wprowadzenie/float-columns.sql

```
ALTER TABLE Bledy ADD COLUMN godziny FLOAT;
ALTER TABLE Konta ADD COLUMN stawka_godzinowa FLOAT;
```

Obie kolumny aktualizujemy na podstawie zapisów w dziennikach prac nad eliminowaniem błędów i stawek godzinowych programistów. Testujemy jeszcze generowanie raportu i jesteśmy gotowi do codziennego przygotowywania potrzebnych zestawień.

Następnego dnia do naszego gabinetu wchodzi szef z kopią raportu o kosztach projektu. „Te liczby nie zgadzają się" — wycedził przez zaciśnięte zęby. „Dla porównania wykonałem te obliczenia ręcznie i okazało się, że Twój raport jest nieprecyzyjny — nieznacznie, różnice wynoszą kilka złotych. Jak to wyjaśnisz?". Jesteśmy przerażeni. Co może nie zgadzać się w tak prostych obliczeniach?

10.1. Cel: stosowanie liczb ułamkowych zamiast liczb całkowitych

Liczba całkowita jest wyjątkowo przydatnym typem danych, ale umożliwia reprezentowanie tylko wartości całkowitoliczbowych, jak 1, 327 czy –19. Ten typ danych nie oferuje możliwości reprezentowania wartości ułamkowych, na przykład 2,5. Jeśli więc potrzebujemy liczb gwarantujących większą precyzję niż liczby całkowite, musimy zastosować inny typ danych. Na przykład kwoty pieniężne zwykle są reprezentowane przez liczby z dwoma miejscami dziesiętnymi, na przykład 19,95 zł.

Naszym celem jest przechowywanie wartości numerycznych niebędących liczbami całkowitymi i wykorzystywanie tych liczb w obliczeniach arytmetycznych. Istnieje jeszcze jeden cel, który jednak jest na tyle oczywisty, że rzadko się o nim wprost mówi — wyniki naszych obliczeń arytmetycznych muszą być **prawidłowe**.

10.2. Antywzorzec: stosowanie typu danych FLOAT

Większość języków programowania obsługuje typy danych dla liczb całkowitych (zwykle pod nazwą `float` lub `double`). Język SQL obsługuje podobny typ danych nazwany `FLOAT`. Wielu programistów odruchowo używa typu

danych `FLOAT` SQL-a wszędzie tam, gdzie potrzebują ułamkowych danych liczbowych, ponieważ są przyzwyczajeni do programowania z wykorzystaniem tego typu danych.

Typ danych `FLOAT` języka SQL, tak jak typ danych `float` w większości języków programowania, koduje liczby rzeczywiste w formacie binarnym zgodnie ze standardem IEEE 754. Efektywne używanie tego typu wymaga zrozumienia wybranych cech liczb zmiennoprzecinkowych reprezentowanych w tym formacie.

Zaokrąglanie z konieczności

Wielu programistów nie zdaje sobie sprawy z istnienia pewnej ważnej cechy tego formatu liczb zmiennoprzecinkowych: nie wszystkie wartości, które możemy zapisywać w formie liczb dziesiętnych, mogą być reprezentowane w formie binarnej. W tej sytuacji niektóre wartości wymagają zaokrąglania do bardzo zbliżonych wartości.

Aby lepiej zrozumieć kontekst tych zaokrągleń, wystarczy przeanalizować przypadek prostej liczby wymiernej, na przykład jednej trzeciej, która jest reprezentowana przez okresowy ułamek dziesiętny 0,333… Prawdziwej wartości nie można reprezentować w formie dziesiętnej, ponieważ musielibyśmy zapisać nieskończoną liczbę cyfr. Liczba cyfr po przecinku to tzw. precyzja reprezentacji odpowiedniej wartości, zatem w pełni prawidłowa reprezentacja ułamka okresowego wymagałaby **nieskończonej precyzji**.

Rozwiązaniem kompromisowym jest stosowanie **skończonej precyzji**, tj. wybór wartości liczbowej możliwie zbliżonej do oryginalnej liczby ułamkowej, na przykład 0,333. Wadą tego rozwiązania jest to, że ta wartość nie jest dokładnie tą samą liczbą, którą chcemy reprezentować.

```
1/3+1/3+1/3 = 1,000
0,333+0,333+0,333 = 0,999
```

Nawet jeśli zwiększymy precyzję, suma trzech przybliżeń jednej trzeciej wciąż nie będzie równa oczekiwanej wartości 1,0. Kompromis polegający na stosowaniu skończonej precyzji dla ułamków okresowych jest jednak niezbędny.

```
1/3+1/3+1/3 = 1,000000
0,333333+0,333333+0,333333 = 0,999999
```

Oznacza to, że niektóre prawidłowe wartości liczbowe, które możemy sobie bez trudu wyobrazić, w ogóle nie mogą być reprezentowane z zastosowaniem metody skończonej precyzji. Część programistów uważa, że

takie rozwiązanie jest usprawiedliwione — skoro wartości złożonych z nieskończonej liczby cyfr i tak nie da się zapisać, każda zapisywana przez nas liczba z natury rzeczy ma skończoną precyzję i tak też powinna być przechowywana w formie binarnej, prawda? Niestety nie.

Zgodnie ze standardem IEEE 754 liczby zmiennoprzecinkowe są reprezentowane w systemie liczbowym o podstawie 2. Oznacza to, że wartości wymagające nieskończonej precyzji w systemie binarnym nie pokrywają się ze zbiorem wartości, które wymagają takiej reprezentacji w systemie dziesiętnym. Niektóre wartości, które wymagają skończonej precyzji w systemie dziesiętnym, na przykład 59,95, wymagają nieskończonej precyzji, jeśli miałyby być dokładnie reprezentowane w systemie binarnym. Typ danych `FLOAT` nie oferuje takich możliwości, zatem stosuje najbliższą obsługiwaną wartość w systemie liczbowym o podstawie 2, czyli wartość odpowiadającą liczbie 59.950000762939 w systemie dziesiętnym.

Niektóre wartości przypadkowo wymagają skończonej precyzji w obu formatach. Jeśli zrozumiemy szczegóły przechowywania liczb w formacie IEEE 754, teoretycznie będziemy potrafili przewidywać, jak poszczególne wartości dziesiętne będą reprezentowane w formacie binarnym. W praktyce jednak większość programistów nie wykonuje podobnych obliczeń dla każdej stosowanej przez siebie liczby zmiennoprzecinkowej. Nie możemy zagwarantować, że kolumna `FLOAT` w bazie danych będzie dostatecznie precyzyjna, zatem nasza aplikacja powinna zakładać, że każda wartość w tej kolumnie mogła zostać zaokrąglona.

Niektóre bazy danych obsługują pokrewne typy danych nazwane `DOUBLE PRECISION` i `REAL`. Precyzja oferowana przez te typy danych i sam typ `FLOAT` zależy co prawda od implementacji bazy danych, ale wszystkie te typy reprezentują wartości zmiennoprzecinkowe ze skończoną liczbą cyfr binarnych, zatem sposób zaokrąglania liczb we wszystkich przypadkach jest podobny.

Stosowanie typu FLOAT w języku SQL

Niektóre bazy danych kompensują wspomnianą niedokładność i wyświetlają właściwe wartości.

Plik Błędy-zaokrągleń/anty/select-rate.sql

```
SELECT stawka_godzinowa FROM Konta WHERE id_konta = 123;
```

Zwraca: `59.95`

Poznać format IEEE 754

Pierwsze propozycje dotyczące standardowego formatu binarnego dla liczb zmiennoprzecinkowych pojawiły się jeszcze w 1979 roku. Specyfikacja, która ostatecznie zyskała status standardu w 1985 roku, jest obecnie powszechnie implementowana w rozmaitych formach oprogramowania, przede wszystkim w językach programowania i mikroprocesorach.

Format składa się z trzech pól niezbędnych do zakodowania wartości zmiennoprzecinkowej: pola **części ułamkowej**, pola **wykładnika**, do którego należy podnieść tę część ułamkową, oraz jednobitowego pola **znaku**.

Jedną z zalet standardu IEEE 754 jest właśnie stosowanie wykładnika, dzięki czemu można w tym formacie reprezentować zarówno bardzo małe, jak i bardzo duże wartości. Format obsługuje nie tylko liczby rzeczywiste, ale też dużo większy przedział wartości niż tradycyjne, stałoprzecinkowe formaty reprezentacji liczb całkowitych. Jeszcze większy przedział reprezentowanych wartości oferuje format podwójnej precyzji. Oznacza to, że opisywane formaty dobrze sprawdzają się w zastosowaniach naukowych.

Bodaj najbardziej popularnym zastosowaniem ułamkowych wartości liczbowych jest reprezentowanie kwot pieniężnych. Stosowanie standardu IEEE 754 dla tego rodzaju wartości nie jest konieczne, ponieważ opisany w tym rozdziale format skalowanych liczb dziesiętnych pozwala równie łatwo i bardziej precyzyjnie obsługiwać kwoty pieniężne.

Dobrymi źródłami wiedzy o tym formacie są artykuł opublikowany na Wikipedii (*http://pl.wikipedia.org/wiki/IEEE_754*) oraz artykuł Davida Goldberga zatytułowany „What Every Computer Scientist Should Know About Floating-Point Arithmetic" [Gol91].

Artykuł Goldberga został też przedrukowany (jest dostępny pod adresem *http://www.validlab.com/goldberg/paper.pdf*).

Rzeczywista wartość przechowywana w kolumnie typu `FLOAT` nie musi jednak odpowiadać tej wartości. Wystarczy pomnożyć tę wartość przez miliard, aby odkryć pewne rozbieżności:

Plik Błędy-zaokrągleń/anty/magnify-rate.sql

```
SELECT stawka_godzinowa * 1000000000 FROM Konta WHERE id_konta = 123;
```

Zwraca: `59950000762. 939`

Można było oczekiwać, że wartość zwrócona przez poprzednie zapytanie po przemnożeniu wyniesie 59950000000,000. Jak widać, wartość 59,95 została zaokrąglona do wartości, która może być reprezentowana w systemie ze skończoną precyzją oferowanym przez format binarny IEEE 754. W tym przypadku różnica jest mniejsza niż jedna milionowa, zatem jej dokładność powinna wystarczyć na potrzeby wielu obliczeń.

Przybliżenie nie gwarantuje jednak wystarczającej dokładności dla innych obliczeń. Dobrym przykładem jest użycie wartości typu FLOAT w wyrażeniu porównującym liczby.

Plik Błędy-zaokrągleń/anty/inexact.sql

```
SELECT * FROM Konta WHERE stawka_godzinowa = 59.95;
```

Wynik: zbiór pusty, brak pasujących wierszy

Jak wiemy, wartość przechowywana w kolumnie stawka_godzinowa w rzeczywistości jest nieznacznie większa niż 59.95. W tej sytuacji, mimo że przypisaliśmy tej kolumnie wartość 59.95 w wierszu z identyfikatorem konta 123, zapytanie nie jest dopasowywane do powyższego zapytania.

Typowym obejściem tego problemu jest traktowanie wartości zmiennoprzecinkowych jako „praktycznie równe", jeśli dzieląca je różnica nie przekracza jakiegoś niewielkiego progu. Wystarczy odjąć jedną wartość od drugiej i użyć dostępnej w języku SQL funkcji wartości bezwzględnej ABS(), aby uzyskana różnica była dodatnia. Jeśli wynik jest równy zero, obie wartości są dokładnie równe. Jeśli wynik jest dostatecznie mały, obie wartości można traktować jako praktycznie równe. Poniższe zapytanie prawidłowo odnajduje interesujący nas wiersz:

Plik Błędy-zaokrągleń/anty/threshold.sql

```
SELECT * FROM Konta WHERE ABS(stawka_godzinowa - 59.95) < 0.000001;
```

Różnica jest jednak na tyle duża, że porównanie z nieco większą precyzją zakończy się niepowodzeniem:

Plik Błędy-zaokrągleń/anty/threshold.sql

```
SELECT * FROM Konta WHERE ABS(stawka_godzinowa - 59.95) < 0.0000001;
```

Dobór właściwego progu zależy od konkretnej liczby, ponieważ wartość bezwzględna różnicy dzielącej wartości dziesiętne od zaokrąglonych reprezentacji binarnych jest różna w przypadku poszczególnych liczb.

Innym przykładem problemu wynikającego z nieprecyzyjnego charakteru typu FLOAT są obliczenia polegające na agregowaniu wielu wartości. Jeśli na przykład użyjemy funkcji SUM() do dodania wszystkich wartości zmiennoprzecinkowych w jakiejś kolumnie, błędy zaokrągleń poszczególnych liczb skumulują się w wyznaczonej sumie.

Plik Błędy-zaokrągleń/anty/cumulative.sql

```
SELECT SUM(b.godziny * k.stawka_godzinowa) AS koszty_projektu
FROM Bledy AS b
JOIN Konta AS k ON (b.przypisany_do = k.id_konta);
```

Skumulowany efekt niedokładności liczb zmiennoprzecinkowych jest jeszcze bardziej widoczny podczas wyznaczania zagregowanego iloczynu zbioru liczb (zamiast ich sumy). Poszczególne różnice wydają się niewielkie, ale z czasem mocno rosną. Jeśli na przykład pomnożymy wartość 1 przez współczynnik równy dokładnie 1,0, wynik zawsze będzie wynosił 1. Nie ma znaczenia, ile razy zastosujemy ten mnożnik. Jeśli jednak użyjemy współczynnika 0,999, wynik będzie inny. Jeśli kolejno pomnożymy liczbę 1 przez wartość 0,999 tysiąc razy, otrzymamy wynik równy około 0,3677. Im więcej operacji mnożenia wykonamy, tym większa będzie ta rozbieżność.

Dobrym przykładem stosowania operacji wielokrotnego mnożenia jest wyznaczanie łącznego oprocentowania na potrzeby kalkulacji finansowych. Używanie niedokładnych liczb zmiennoprzecinkowych powoduje błędy, które początkowo wydają się zupełnie niegroźne, ale z czasem, skumulowane, zaczynają stwarzać poważne problemy. Stosowanie precyzyjnych wartości w aplikacjach finansowych jest więc bardzo ważne.

10.3. Jak rozpoznać ten antywzorzec

Niemal każde użycie typów danych FLOAT, REAL lub DOUBLE PRECISION jest podejrzane. Większość aplikacji korzystających z liczb zmiennoprzecinkowych w rzeczywistości nie potrzebuje przedziału wartości obsługiwanego przez formaty zgodne ze standardem IEEE 754.

Korzystanie z typów danych FLOAT w języku SQL wydaje się o tyle naturalne, że podobny typ (często nawet pod tą samą nazwą) występuje w większości języków programowania. Okazuje się jednak, że można wybrać lepszy typ danych.

10.4. Usprawiedliwione użycia tego antywzorca

FLOAT jest dobrym typem danych w sytuacji, gdy potrzebujemy liczb rzeczywistych z przedziału większego niż ten obsługiwany przez typy INTEGER i NUMERIC. Naukowe aplikacje często wskazuje się jako przykład uzasadnionego stosowania typu FLOAT.

W systemie Oracle typu danych FLOAT używa się do wyrażania dokładnie skalowanych wartości liczbowych, zaś do reprezentowania niedokładnych wartości numerycznych stosuje się typ danych BINARY_FLOAT (zgodny ze standardem kodowania IEEE 754).

10.5. Rozwiązanie: stosowanie typu danych NUMERIC

Zamiast typu FLOAT i typów pokrewnych możemy stosować typy danych NUMERIC lub DECIMAL języka SQL dla liczb ułamkowych stałej precyzji.

Plik Błędy-zaokrągleń/roz/numeric-columns.sql

```
ALTER TABLE Bledy ADD COLUMN godziny NUMERIC(9,2);

ALTER TABLE Konta ADD COLUMN stawka_godzinowa NUMERIC(9,2);
```

Wymienione typy danych umożliwiają dokładne reprezentowanie wartości numerycznych z maksymalną precyzją określoną podczas definiowania odpowiednich kolumn. Precyzję należy określić w formie argumentu typu danych — obowiązująca składnia przypomina trochę sposób określania długości typu danych VARCHAR. Precyzja to łączna liczba cyfr dziesiętnych, których możemy używać dla wartości w tak zdefiniowanej kolumnie. Precyzja równa 9 oznacza, że możemy przechowywać takie wartości jak 123456789, ale najprawdopodobniej nie będziemy mogli obsłużyć wartości równej 1234567890.[1]

[1] W niektórych systemach baz danych rozmiar tej kolumny jest zaokrąglany w górę do najbliższego bajta, słowa lub podwójnego słowa, zatem w pewnych przypadkach maksymalna wartość w kolumnie typu NUMERIC może składać się z większej liczby cyfr, niż to wynika ze wskazanej precyzji.

Istnieje też możliwość określenia skali za pośrednictwem drugiego argumentu tego typu danych. Skala decyduje o liczbie cyfr na prawo od przecinka oddzielającego część całkowitą od ułamkowej. Cyfry skali są odliczane od liczby cyfr precyzji, zatem precyzja 9 ze skalą 2 oznaczają, że możemy przechowywać takie wartości jak `1234567.89`, ale już nie wartości `12345678.91` czy `123456.789`.

Określane przez nas precyzja i skala są stosowane dla danej kolumny we wszystkich wierszach tabeli. Innymi słowy, nie możemy przechowywać wartości ze skalą 2 w części wierszy i wartości ze skalą 4 w pozostałych. W języku SQL to naturalne, że typ danych kolumny jest konsekwentnie stosowany dla wszystkich wierszy (tak jak w przypadku kolumny typu `VARCHAR(20)`, gdzie każdy wiersz może zawierać łańcuch określonej długości).

Zaletą typów `NUMERIC` i `DECIMAL` jest możliwość przechowywania liczb wymiernych bez ryzyka ich zaokrąglania (jak w przypadku typu `FLOAT`). Po zapisaniu wartości `59.95` możemy być pewni, że w bazie danych jest przechowywana dokładnie ta liczba. Jeśli porównamy ją ze stałą wartością `59.95`, okaże się, że obie wartości są sobie równe.

Plik Błędy-zaokrągleń/roz/exact.sql

```
SELECT stawka_godzinowa FROM Konta WHERE stawka_godzinowa = 59.95;
```

Zwraca: `59.95`

Podobnie, jeśli pomnożymy tę wartość przez miliard, otrzymamy oczekiwaną wartość:

Plik Błędy-zaokrągleń/roz/magnify-rate-exact.sql

```
SELECT stawka_godzinowa * 1000000000 FROM Konta WHERE stawka_godzinowa = 59.95;
```

Zwraca: `59950000000`

Typy danych `NUMERIC` i `DECIMAL` zachowują się identycznie; nie powinny występować żadne różnice w ich działaniu. Istnieje też synonim `DEC` dla typu danych `DECIMAL`.

Nadal nie możemy przechowywać wartości wymagających nieskończonej precyzji, na przykład jednej trzeciej. Proponowane typy umożliwiają nam jednak przechowywanie dokładnej reprezentacji liczb w formie, w której zapisujemy je w systemie dziesiętnym.

Jeśli więc potrzebujemy dokładnych wartości dziesiętnych, powinniśmy posługiwać się typem danych `NUMERIC`. Typ danych `FLOAT` nie może reprezentować wielu dziesiętnych liczb wymiernych, zatem wartości tego typu należy zawsze traktować jako niedokładne.

> Jeśli tylko możemy tego uniknąć, nie powinniśmy używać typu `FLOAT`.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Ucz się na błędach... Cudzych!

Podobno najlepiej jest uczyć się na cudzych błędach. Ta mądrość sprawdza się w wielu dziedzinach życia, także w informatyce. Ile razy brnąłeś w złe rozwiązanie, aż ktoś powiedział: „Tak się nie robi"? Czy nie pomyślałeś wtedy o książce, w której zamiast metod rozwiązywania danych problemów znalazłbyś informacje, jak do nich nie dopuścić? Chciałbyś, żeby ta książka dotyczyła języka SQL? Oto ona!

Niniejsza publikacja przedstawia zbiór antywzorców w języku SQL. Dzięki niej poznasz błędy najczęściej popełniane przy projektowaniu i wykorzystywaniu baz danych. Dowiesz się z niej, jak nie tworzyć logicznego i fizycznego projektu bazy danych, jak nie zadawać zapytań SQL oraz jak nie wytwarzać aplikacji — a wszystko po to, aby zrozumieć, jak nie popełniać błędów. W każdej z czterech części znajdziesz ogrom interesujących informacji: poznasz zasady przechowywania haseł, błędy pojawiające się w wyniku zaokrągleń czy sposoby radzenia sobie z brakiem integralności bazy danych. Książka ta jest genialną pozycją, dzięki której już nigdy nie zboczysz z drogi w codziennej pracy z bazami danych i językiem SQL!

- Antywzorce logicznego projektu bazy danych
  - Modelowanie drzew
  - Tworzenie kluczy głównych
  - Związki polimorficzne
  - Atrybuty wielokolumnowe
- Antywzorce fizycznego projektu bazy danych
  - Błędy zaokrągleń
  - Przechowywanie dużych plików
  - Indeksy
- Antywzorce zapytań
  - Wykorzystanie NULL-a
  - Grupowanie kolumn
  - Losowe wybieranie wiersza
  - Przeszukiwanie tekstów
  - Optymalizacja zapytań SQL
- Antywzorce wytwarzania aplikacji
  - Przechowywanie haseł
  - Wstrzykiwanie SQL

Pragmatic Bookshelf

helion.pl księgarnia internetowa

Nr katalogowy: 7514

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Informatyka w najlepszym wydaniu

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena: 69,00 zł

ISBN 978-83-246-3482-8

9 788324 634828

sięgnij po WIĘCEJ
KOD KORZYŚCI