Krzysztof Stencel
Instytut Informatyki
Uniwersytet Warszawski

Homo informaticus colligens, czyli człowiek zbierający dane

Dane zgromadzone przez przedstawicieli *Homo informaticus* są niewyobrażalne. Mierzy się je setkami eksabajtów. Jeden eksabajt to aż 260 bajtów, czyli około 1 tryliona bajtów (1018). *Colligo, ergo sum* (czyli gromadzę więc jestem) to jedno z ulubionych powiedzonek niektórych *Homo informaticus*, którzy stanowią istotny podgatunek zwany *Homo informaticus colligens*. Aby *Homo informaticus* mógł robić to, co umie najlepiej, czyli przetwarzać dane, ktoś musi tymi danymi zarządzać, przechowywać je oraz zabezpieczać je przed niepowołanym dostępem. Tym właśnie zajmuje się jego bliski ziomek – *Homo informaticus colligens* (znany także jako *Infocolligens*), o którym jest ta opowieść.

W tym rozdziale zajmiemy się opisem zadań, wyzwań i metod działania *Infocolligensa*. Do gromadzenia i udostępniania danych używa on systemów baz danych. Omówimy różne rodzaje baz danych, od historycznych (choć wciąż jeszcze używanych) do obecnie powszechnie stosowanych, aż po nowinki, które już zadomowiły się w gawrze *Infocolligensa*. Przedstawimy koncepcje baz sieciowych, hierarchicznych i obiektowych, których czas popularności już minął. Zajmiemy się dokładniej obecnie najpowszechniej stosowanymi bazami relacyjnymi i ich podstawowym językiem SQL. Przekonamy się, że mimo całej złożoności, jest to język bardzo użyteczny i ostatecznie nie tak trudny do opanowania. Na końcu opiszemy bazy nurtu NOSQL, które przebojem wdarły się do wielkich firm i mają obecnie bardzo wiele zastosowań: bazy klucz-wartość, dokumentowe i grafowe. Dynamika rozwoju dziedziny baz danych zapiera obecnie dech w piersiach. Może i Ty chcesz wspomóc *Infocolligensa*?

1. Homo informaticus colligens

Homo informaticus będący przedmiotem niniejszego tomu jest silnie uzależniony od symbiozy z pewną odmianą swojego gatunku, egzystującą cichutko na skraju świata technologii informacyjnych (IT). Ową tajemniczą odmianą jest *Homo informaticus colligens*[1], *Homo informaticus* gromadzący (dane). Czymże bowiem zajmuje się *Homo informaticus*? Wykorzystuje lub buduje algorytmy przetwarzające dane. Mogą to być dane (wejściowe), które skądś trzeba pobrać, lub wyniki (dane wyjściowe), które gdzieś należy odesłać w celu wykorzystania lub przechowania do przyszłego użycia. Jakkolwiek można sobie wyobrazić algorytm, który nie potrzebuje danych (np. generator liczb losowych), to algorytm nigdy nieprodukujący żadnych wyników[2] jest całkowicie bezużyteczny. Dane przetwarzane przez *Homo informaticus* trzeba więc dostarczyć, odebrać i przechować. W tym miejscu widać wyraźnie rolę *Homo informaticus colligens* (w skrócie: *Infocolligens*). To właśnie on w złożonym ekosystemie świata technologii informacyjnych zajmuje się składowaniem i udostępnianiem danych.

Ilość danych, jakimi obecnie dysponuje ludzkość, mierzy się setkami eksabajtow. Jeden eksabajt to 2^{60} bajtów, czyli 1 152 921 504 606 846 976, tj. około tryliona bajtów (10^{18} = 1 000 000 000 000 000 000). *Colligo, ergo sum* (pl. gromadzę więc jestem) – to na pewno maksyma wielkich korporacji informatycznych, dysponujących tak ogromnymi zbiorami danych. Głównymi zasobami takich firm jak Google, Facebook, Amazon itd. nie jest już wcale oprogramowanie. O ich pozycji świadczy ilość i jakość zgromadzonych przez nie danych oraz doskonałość metod ich przetwarzania.

W ekosystemach przyrodniczych nadmierne rozmnożenie jednego gatunku stanowi nie lada problem. Zasoby pożywienia są ograniczone, więc równowaga ekosystemu jest po pewnym czasie mniej lub bardziej brutalnie przywracana. O dziwo, tej właściwości nie ma ekosystem IT. Nowi *Homines informatici* zasiedlający ten ekosystem produkują takie ilości nowych danych, że nie może być mowy o ich deficycie. Zadaniem *Infocolligensa* jest znaleźć sposób przechowywania tej rosnącej wciąż liczby. Gawrę *Infocolligensa* zwykliśmy nazywać **bazą danych**. Tam właśnie gromadzi on dane i stamtąd udostępnia je użytkownikom.

[1] Określenie *Homo colligens* zostało zaczerpnięte z książki Claire Chavarin, *Homo colligens: essai sur l'imaginaire de la collection au 19e siecle*, 2001.

[2] Wyniki mogą mieć rozmaitą postać – na przykład informacji ostrzegawczych z oprogramowania monitorującego pewne urządzenie. Gdy urządzenie pracuje poprawnie, żadnych ostrzeżeń nie ma. Wyniki pojawią się, gdy ulegnie ono awarii.

2. Bazy danych

Zawartość i struktura baz danych[3] zmieniała się rozmaicie na przestrzeni ostatnich kilkudziesięciu lat. Jedno pozostało jednak niezmienione – pojęcie **rekordu**, jako podstawowej jednostki danych. Rekord był, jest i zapewne będzie kolekcją **pól**. Każde pole ma nazwę i wartość. W rozmaitych modelach baz danych zmieniały się wymagania odnośnie rekordów co do nazw pól (czy pola w rekordzie mogą się powtarzać?) i typów danych umieszczanych w polach (jak złożone mogą być, czy tylko proste, a może też złożone?). Zmieniały się także dodatkowe cechy, którymi mógł być obdarzony rekord (np. pojęcie sąsiedztwa z innymi rekordami, które służyło nałożeniu na rekordy struktury grafowej). Poszczególne elementy znikały i pojawiały się zależnie od potrzeb i mód. Rekord pozostał jednak zawsze tym samym – zestawem pól. Pod tym względem *Info-colligens* nie zmienił się ani na jotę.

W pierwszych bazach danych, tzw. **sieciowych** i **hierarchicznych**, rekordy mogły być powiązane w sieci lub hierarchie. Każdy rekord oprócz wartości swoich pól (np. rekord z danymi osobowymi zawierał pola, takie jak nazwisko, imię, data urodzenia) mógł mieć także zdefiniowanych sąsiadów (następniki). Przykładowo rekord z danymi osobowymi Kowalskiego był połączony krawędzią z rekordem działu, w którym pracował, czy z rekordem jego szefa. W modelu hierarchicznym posługiwano się pojęciem rekordu podrzędnego (potomnego), a rekordy łączono w hierarchie. Poszczególne rekordy mogą jednak występować w kilku hierarchiach, więc żeby móc je zapisać w hierarchicznej bazie danych, dodano do jej modelu pojęcie **rekordu wirtualnego** znajdującego się w liściu hierarchii. Taki rekord był po prostu traktowany jako odwołanie do dowolnego innego rekordu.

Rekordy o tych samych właściwościach (np. dane osobowe, działy i samochody służbowe) były przechowywane w plikach. **Plik** jest jednocześnie jednostką organizacyjną składowiska fizycznego – dysku lub taśmy. Z bazy danych korzystano metodą „jedna operacja – jeden rekord". Interfejs bazy danych przewidywał następujące instrukcje: weź pierwszy rekord z pliku, weź następny rekord z pliku, weź pierwszego sąsiada (w bazie hierarchicznej było to dziecko), weź kolejnego sąsiada (dziecko). Posługiwano się także indeksami, z tym że operacje były równie jednorekordowe: weź pierwszy rekord o danej wartości klucza wyszukiwania, weź kolejny itd.

[3] Spośród książek dostępnych w języku polskim najwięcej informacji o bazach danych można znaleźć w klasycznym podręczniku [2].

IMS (ang. *Information Management System*) firmy IBM to jeden z pierwszych hierarchicznych systemów baz danych. Prace nad nim rozpoczęto w roku 1966, a pierwszą wersję uruchomiono w 1968 roku. System ten nadal jest pielęgnowany i użytkowany. Dlaczego nie? Przecież działa. Należy pamiętać, że lepsze jest wrogiem dobrego.

Przełom zaczął się w roku 1970, gdy Edgar Frank Codd przedstawił relacyjny model danych. Zaproponował on uproszczenie rekordów poprzez likwidację struktury grafowej ponad nimi. Okazało się, że mniej znaczy więcej. Dzięki temu uproszczeniu można było zdefiniować nowy abstrakcyjny interfejs do bazy danych działający na zasadzie jedna operacja – jedna relacja. **Relacja** to nic innego jak kolekcja takich uproszczonych rekordów. Operowanie na relacjach, a nie na pojedynczych rekordach było strzałem w dziesiątkę, m.in. ze względu na właściwości dysków magnetycznych. Wirujące dyski magnetyczne mają bowiem to do siebie, że odczyt pojedynczego sektora zajmuje niemal tyle samo czasu co odczyt całego cylindra[4]. Przetwarzanie danych całymi relacjami (tj. dużymi zbiorami rekordów) umożliwia wykorzystanie tej właściwość dysków. Od roku 1970 *Homo informaticus colligens* ma nad biurkiem transparent *Interfejs, głupcze!* Przemysł zaś już od roku 1986 przyjął oficjalny standard języka zapytań dla relacyjnych baz danych. Jest nim **SQL** (ang. *Structured Query Language*), o którym więcej piszemy w dalszej części tego rozdziału. SQL to właśnie język przetwarzania relacji, a nie rekordów.

Pierwszy relacyjny system zarządzania bazą danych (SZBD) powstał też w laboratoriach IBM w roku 1973. Dalej w tym rozdziale znajdują się informacje o jego optymalizatorze zapytań. Algorytmy zaimplementowane w tym optymalizatorze nadal są wykorzystywane we współczesnych bazach danych. Pierwszy komercyjny system relacyjny zbudowano w roku 1979 w firmie Relational Software, która obecnie nosi nazwę Oracle.

Z kolei nad drzwiami *Infocolligensa* wisi napis *Abstrakcja, głupcze!* Pojawienie się relacyjnych baz danych poskutkowało wprowadzeniem **fizycznej niezależ-**

[4] Cylinder składa się ze wszystkich sektorów wszystkich ścieżek o takim samym rzucie na podstawę urządzenia dyskowego.

ności danych. Struktury fizyczne (pliki) zostały oderwane od struktur logicznych (relacji). Projektant bazy danych tworzy jej schemat w postaci zestawu relacji; programista aplikacyjny implementuje aplikację odwołując się do relacji. Pliki natomiast są dla niego niewidoczne. Nie zna ich położenia (mogą być nawet na zdalnej maszynie) oraz przypisania plików do relacji. Dzięki temu projektant bazy danych, a później jej administrator, może przenosić dane z jednego fizycznego położenia do innego bez wpływu na działające aplikacje.

Wspomniane powyżej oddzielenie aspektów fizycznych od logicznych ma jeszcze jedną kluczową konsekwencję. Jest nią możliwość zdefiniowania języka zapytań abstrahującego od fizycznych struktur składowania i konkretnych algorytmów przetwarzania danych. O tym wspominamy w następnym punkcie.

Pierwszą wersję języka SQL opublikowano w roku 1986, a następne wersje pojawiały się w latach 1989, 1992, 1999, 2003, 2006 i 2008. Część z nich zawierała przełomowe zmiany. Wersja SQL:1999 to już pełny język programowania, w których można zapisać dowolny algorytm, wcześniej ten język miał istotne ograniczenia siły wyrazu. Ta wersja to też pierwszy standard obiektowo-relacyjny. Z kolei w języku SQL:2003 uwzględniono obsługę danych w formacie XML.

2.1. Obiektowość w bazach danych

Wzrost popularności programowania obiektowego sprawił, że zaczęto zastanawiać się nad nowym modelem danych. Skoro aplikacje przetwarzają obiekty, to dlaczego baza danych ma składować krotki relacji? Odpowiedzią na to pytanie było pojawienie się **obiektowych baz danych**. Rekord w takiej bazie jest obiektem, który może mieć złożone typy pól oraz pola referencyjne odwołujące się do innych obiektów. Tak jak w programowaniu obiektowym, obiekty-rekordy mogą mieć metody. W latach 90. XX wieku powstały liczne obiektowe systemy zarządzania bazami danych (**OSZBD**), a nawet pierwsza propozycja standardu wydana przez konsorcjum ODMG (rozwiązane w roku 2001). Po kilkunastu latach rozwoju baz obiektowych zainteresowanie nimi świata naukowego osłabło i nie zdobyły one dużego udziału w rynku. Przyczyn doszukiwano się rozmaitych. Mogły nimi być słabość kapitałowa i technologiczna firm wytwarzających OSZBD, słabość merytoryczna konsorcjum standaryzacyjnego czy mniejsza od oczekiwań wygoda stosowania OSZBD. Przecież miały one ułatwić życie programistom, więc dlaczego były o niebo mniej przyjazne niż systemy relacyjne? Nie bez zna-

czenia jest też duże przywiązanie obiektowych produktów bazodanowych do konkretnych języków programowania.

W roku 2006 konsorcjum OMG (ang. *Object Management Group*, m.in. właściciele standardu UML) próbowało ożywić koncepcję standardu obiektowych baz danych. Powołano odpowiednią grupę roboczą. Zebrano ze świata propozycje i po kilku spotkaniach zdecydowano się przyjąć **architekturę stosową** Kazimierza Subiety [4] za punkt wyjścia do definicji nowego standardu. Ten duży sukces polskiej koncepcji niestety spotkał się z bojkotem wielkim firm informatycznych i niechęcią środowiska programistów, jako coś zupełnie nowego i zupełnie nieznanego. Grupa robocza przestała się spotykać i obecnie słuch o niej zaginął.

Reakcja wytwórców relacyjnych SZBD miała dodatkowe ostrze. Była nim koncepcja obiektowo-relacyjnego modelu danych oraz odwzorowania obiektowo-relacyjne. Nowy model danych miał na celu pogodzenie ognia z wodą, bo oba modele mają raczej mało wspólnego ze sobą. Wynikiem jest m.in. nowy standard SQL. Począwszy od SQL:1999, ten podstawowy język programowania baz danych nie jest już relacyjny, a obiektowo-relacyjny. Wtręty obiektowe do SQL okazały się umiarkowanym sukcesem i nie są zbyt powszechnie używane. Z drugiej strony systemy odwzorowania obiektowo-relacyjnego (np. system Hibernate) podbiły umysły wielu programistów i architektów oprogramowania. Systemy takie umożliwiają tworzenie aplikacji obiektowo, zakładając że wskazane obiekty będą trwale składowane w jakiejś bazie danych. Dostarczają także mechanizmów odwzorowania danych obiektowych na relacje bazy danych. Programista definiuje więc tylko obiekty, które chce mieć składowane w bazie, a system odwzorowania martwi się już o wszystko inne. Hibernate i inne systemy tego typu sprawiają, że obiektowa baza danych nie wydaje się już ani trochę atrakcyjna.

Na przełomie wieków XX i XXI wydawało się, że to już koniec historii rozwoju baz danych. Relacyjne bazy danych miały być narzędziem do składowania i przetwarzania wszelkich danych. Kto tak myślał, pomylił się tak samo jak Francis Fukuyama ogłaszający koniec historii w polityce. Jego teorie runęły razem z wieżami WTC tragicznego 11 września 2001 roku. Niestety mniej więcej w tym samym czasie na przełomie wieków doszło też do dramatycznych zmian w dziedzinie baz danych. Okazało się, że relacyjne bazy danych i SQL nie rozwiązują wszystkich problemów. Największym z nich była skalowalność. W roku 2000 Eric Brewer sformułował tzw. hipotezę CAP (ang. *Consistency-Availability--Partition tollerance*), według której nie jest możliwe uzyskanie naraz spójności danych (C), pełnej dostępności (A) i poprawności partycjonowania (P). Możliwe są kombinacje jedynie dwuliterowe. Hipoteza Brewera została udowodniona w roku 2002 i nosi teraz zasłużone miano **twierdzenia CAP**.

Konsekwencją potrzeb *Homo informaticus colligens* oraz prawdziwości CAP jest nowy prąd w dziedzinie baz danych, tzw. **ruch NOSQL**. Należy go rozumieć, jako *Not-Only-SQL* (nie tylko SQL), a nie *No-SQL* (tylko nie SQL!), ponieważ w wielu zastosowaniach klasyczne relacyjne bazy danych są nadal najlepszym narzędziem. Najbardziej znaczącymi systemami baz danych NOSQL są: Big-Table, Cassandra czy HBase. W ten sposób *Infocolligens* przeszedł od ery *one size fits all* (do wszystkiego jest dobry jeden „rozmiar" – relacyjna baza danych) do ery *right tool for the job* (do każdego zadania wybierz najlepsze narzędzie: bazę relacyjną lub NoSQL), [3].

Konsorcjum ODMG zawiązano w 1991 roku. W czasie swojej działalności ODMG wydało pięć wersji standardu dla obiektowych baz danych. Niestety nigdy nie stał się on równoważnikiem SQL i zebrał wiele krytycznych ocen. W roku 2001 konsorcjum ogłosiło swój sukces i zostało rozwiązane, gdyż po takim sukcesie nie pozostało już nic do roboty. Sukces był niestety wyłącznie marketingowy, bo standardy ODMG nigdy nie zostały szeroko rozpowszechnione ani zaimplementowane.

3. Relacyjne bazy danych

Od wielu lat gawrę *Infocolligensa* wypełnia głównie serwer relacyjnej bazy danych, chociaż, jak już wiemy, nowe elementy zaczynają się też tam pojawiać. Relacyjna bazy danych jest oparta na pojęciu relacji, pochodzącym z teorii zbiorów (teorii mnogości). Szczęśliwie można jednak objaśnić jej działanie bez odwoływania się do matematyki. Zaczniemy od tego, że zamiast o relacjach będziemy mówić o **tabelach**. Relacyjna baza danych przechowuje pewną liczbę nazwanych tabel. Każda tabela ma stałą liczbę nazwanych kolumn. Dla każdej kolumny jest określony prosty typ danych, które można w nich umieścić (np. napis, liczba, data). Zestaw nazw tabel, nazw ich kolumn oraz ich typów nazywamy **schematem** bazy danych. Dane mają więc postać wierszy w tabelach i w każdym wierszu jest podana wartość w każdej kolumnie[5].

[5] W standardzie SQL tą wartością może być NULL – wartość pusta lub nieokreślona. Ma ona masę paskudnych właściwości włącznie z tym, że jej pojawienie się wprowadza nas w świat logiki trójwartościowej. NULL to „nie-wiadomo-co", więc porównanie czegokolwiek z NULL daje nam trzecią wartość logiczną „nie-wiadomo-czy" – ani fałsz, ani prawda. W tym rozdziale zbywamy NULL milczeniem tak, jak na to zasługuje.

Osoby			
Id	Imię	Nazwisko	Wiek
1	Adam	Stawski	17
2	Lech	Boruta	13
3	Anna	Welke	14
4	Agnieszka	Nguyen	14
5	Teodor	Markowetz	15

Hobby			
IdOsoby	Nazwa	Początek	Koniec
1	Szachy	2000	9999
1	Brydż	2007	9999
1	Wyścigi	2003	2005
3	Szachy	2009	9999
3	Brydż	1999	2008
4	Filatelistyka	1987	9999
c5	Filatelistyka	1987	1994
5	Brydż	1991	9999
5	Filatelistyka	2000	9999

Znajomi				
IdOsoby	IdZnajomego	Sposób	Początek	Koniec
1	2	kolega	1998	9999
2	1	kolega	1998	9999
1	3	ziom	2004	9999
3	1	ziom	2004	9999
1	5	kolega	2011	9999
2	4	kolega	2002	2007
4	5	przyjaciel	2003	2005
5	4	przyjaciel	2003	2005

Rysunek 1. Trzy przykładowe tabele bazy danych dla sieci społecznościowej

Na rysunku 1 pokazano przykład uproszczonej bazy danych dla sieci społecznościowej. Jeśli podczas lektury na temat historii baz danych zastanawiałeś się, co stało się z zależnościami między rekordami, to teraz masz już odpowiedź. Relacyjna baza danych nie oznacza, że nie można zapisać informacji o takich zależnościach. Po prostu są one reprezentowane inaczej. Tabele mogą (a nawet powinny) mieć tzw. **klucz główny**, czyli jedną kolumnę (lub ich zestaw) jednoznacznie identyfikującą każdy rekord. Klucz w tabeli `Osoby` składa się jedynie z kolumny `Id`. W dwóch pozostałych tabelach rzecz jest bardziej złożona. Klucz główny tabeli `Hobby` musi składać się aż z trzech kolumn: `IdOsoby`, `Nazwa` i `Początek`. Wydawać by się mogło, że wystarczą tylko dwie pierwsze kolumny. To jednak nie wystarczy, może się bowiem zdarzyć, że ktoś porzuci pewnego dnia swoje hobby, aby po latach do niego wrócić. Będzie miał wtedy w tabeli `Hobby` dwa rekordy ze swoim identyfikatorem i nazwą zainteresowania. Nie byłoby to możliwe przy zbyt ubogim kluczu głównym. W bazie ma rysunku 1 nie mogłyby więc istnieć dwa rekordy określające zainteresowania Teodora Markowetza (`Id = 5`) filatelistyką. Kolumna `Początek` musi być zatem dodatkowym elementem klucza głównego. Podobnie jest z tabelą `Znajomi`. Tu też może się zdarzyć, że ktoś kogoś przestanie lubić, a potem polubi ponownie. Lekarstwem na to jest znów dodanie kolumny `Początek` do klucza głównego, który w tej tabeli będzie składał się jeszcze z kolumn `IdOsoby` i `IdZnajomego`.

Tabele z rysunku 1 są ilustracją możliwości przechowywania **danych temporalnych**, tzn. takich, które zmieniają się w czasie. Każdy fakt w tabelach `Hobby` i `Znajomi` ma przypisany okres swojego obowiązywania. Zauważmy, że sprytnie uniknęliśmy konieczności używania wartości pustej. Jeśli jakiś fakt jest nadal prawdą, to za koniec jego ważności uznajemy dostatecznie odległą datę. Przy dzisiejszym postępie technologicznym możemy być pewni, że nasza baza danych nie przetrwa najbliższych dwudziestu lat bez gruntownej modernizacji, nie mówiąc już o roku 9999[6].

Przyjrzymy się teraz możliwościom przetwarzania danych, jakie dają relacyjne bazy danych i wspomniany już język SQL na kilku przykładach zapytań. W założeniach, język SQL miał być przyjaznym i podobnym do naturalnego sposobem zapisywania tego, *co* ma być wynikiem zapytania, a nie *jak* ten wynik należy obliczyć. W jakim stopniu SQL spełnia pokładane w nim nadzieje pozostawiamy do indywidualnej oceny. Osobiście, autor ma wątpliwości co do jego przyjazności. Udało się natomiast doskonale oderwać w SQL realizację zapytań od konkretnych algorytmów. Nawet te najprostsze zapytania mają w SQL po kilka możliwych sposobów realizacji, a te bardziej złożone mają tak

[6] Warto pamiętać o skali czasu. Tak zwana pluskwa milenijna w roku 2000 nie wyrządziła żadnych szkód, podczas gdy sekunda przestępna dodana do zegarów ostatniego dnia czerwca 2012 roku spowodowała awarię licznych serwisów internetowych.

wiele realizacji, że wybór jednej właściwej staje się problemem. Zajmiemy się tym w punkcie 5.

Rozważmy pierwsze zapytanie, które ma wypisać wszystkie kolumny wierszy tabeli Osoby, w których w kolumnie Wiek jest wartość 14. Klauzula SELECT wskazuje kolumny, które mają być wypisane. Gwiazdka w klauzuli jest żądaniem wypisania wszystkich kolumn. Klauzula FROM określa, z których tabel należy pobrać wiersze, a WHERE – zawiera warunek filtrujący wiersze. Zostaną wypisane tylko te wiersze tabeli, które spełniają formułę z klauzuli WHERE. Prawda, że proste?

```
SELECT *
FROM Osoby
WHERE Wiek = 14;
```

A gdybyśmy chcieli wypisać imiona i nazwiska osób mających tyle samo lat co osoba nazwiskiem Nguyen? Proszę bardzo! SQL jest kompozycyjny. Zamiast liczby 14 wystarczy wstawić zapytanie obliczające identyfikator odpowiedniej osoby[7]:

```
SELECT o.Imię, o.Nazwisko
FROM Osoby o
WHERE Wiek = (SELECT Id FROM Osoby WHERE Nazwisko = 'Nguyen');
```

Możemy też zapytać o obecne (tj. w roku bieżącym) hobby każdej osoby. W tym celu trzeba pobrać dane z kilku tabel naraz. Aby tego dokonać, należy w klauzuli FROM podać tabele będące źródłami danych. Zapytanie takie wypisuje wynik tak, jakby brało pod uwagę wszystkie możliwe *n*-tki (w tym przykładzie: pary) wierszy z podanych tabel i następnie filtrowało je pozostawiając tylko te, które spełniają warunek WHERE. Wygląda to więc tak, że bierzemy pod uwagę wszystkie możliwe pary osoba-hobby i potem pozostawiamy tylko te, w których między wartościami kolumn Początek i Koniec mieści się 2012 oraz mają takie same wartości w kolumnach Id w tabeli Osoba i IdOsoby w Hobby.

```
SELECT o.Imię, o.Nazwisko, h.Nazwa
FROM Osoby o, Hobby h
WHERE o.Id = h.IdOsoby
   AND 2012 BETWEEN h.Początek AND h.Koniec;
```

[7] Oczywiście, jeśli osób o nazwisku Nguyen jest w tabeli więcej, to wykonanie zapytania zakończy się sygnalizacją błędu. Aby tego uniknąć, można np. wziąć minimalny identyfikator wśród wszystkich tych osób MIN(Id).

Na potrzeby tego zapytania, w klauzuli `FROM` wprowadzono **aliasy** `o` i `h`, zwane też **zmiennymi krotkowymi**, czyli nowe nazwy tabel `Osoby` i `Hobby`. W ramach tego zapytania należy posługiwać się już tylko nazwami `o` i `h`. Można by nie wprowadzać aliasów, jednak ze względu na ewentualne przyszłe zmiany zestawu kolumn w tabelach (tzw. **ewolucję schematu**), zaleca się programistom aplikacyjnym aliasowanie wszystkich tabel.

Obliczenie wyniku zapytania przez system zarządzania bazą danych oczywiście nie polega na wyznaczeniu najpierw wszystkich możliwych par wierszy tych dwóch tabel występujących w zapytaniu (czyli ich iloczynu kartezjańskiego). Dlaczego „oczywiście"? Czasowa złożoność obliczeniowa takiego algorytmu byłaby bowiem funkcją kwadratową, a dokładnie, byłaby iloczynem wielkości obu tabel. Złożoność kwadratowa jest jednak jednym z największych wrogów efektywności obliczeń prowadzonych na bazach danych. Algorytmy stosowane w mechanizmach baz danych muszą mieć złożoność linową ($O(n)$) lub ewentualnie liniowo-logarytmiczną ($O(n\log n)$), gdzie n jest łączną liczbą wierszu w przetwarzanych tabelach. Aparat wykonawczy zapytań omówiono w punkcie 5.

Język SQL umożliwia też wyliczanie pewnych podsumowań. Oto zapytanie, które dla każdej osoby wyznacza liczbę jej rożnych hobby, uwzględniając także chwilowe zainteresowania. W poniższym zapytaniu jest wykonywane standardowe, poznane wcześniej złączenie tabel `Osoby` i `Hobby`, a następnie grupowanie `GROUP BY` par wierszy według wartości kolumn `Id`, `Imię` i `Nazwisko`. W jednej grupie wszystkie pary wierszy mają te same wartości kolumn grupujących. Jedna grupa w tym zapytaniu odpowiada jednej osobie. Następnie w ramach każdej grupy wyznaczana jest liczba rożnych (`DISTINCT`) wartości kolumny `Nazwa` z tabeli `Hobby`. Przed obliczeniem tego podsumowania należy usunąć duplikaty wśród nazw hobby, ponieważ ktoś mógł mieć jakieś hobby, następnie je porzucić i później do niego wrócić. W takim przypadku będzie miał więcej wpisów w tabeli `Hobby` dla jednego swojego zainteresowania, a zatem bez usunięcia duplikatów wynik zapytania byłby niepoprawny. Na końcu zapytania wynik jest sortowany malejąco (`ORDER BY... DESC`) względem liczby różnych hobby, tj. kolumny wyniku `Z`:

```
SELECT o.Id, o.Imię, o.Nazwisko, COUNT(DISTINCT h.Nazwa) Z
FROM Osoby o, Hobby h
WHERE o.Id = h.IdOsoby
GROUP BY o.Id, o.Imię, o.Nazwisko
ORDER BY Z DESC;
```

Zajmijmy się teraz społecznościowym aspektem bazy danych z rysunku 1. Baza ta dopuszcza asymetrię znajomości, tzn. `Adam` może znać `Teodora`, podczas

gdy Teodor może nie znać Adama lub się do tego nie przyznaje. Poniżej znajduje się zapytanie, którego celem jest wyszukanie osób mających obecnie (tj. w 2012) znajomych, którzy nie są ich znajomymi. Tym razem skorzystamy z tabeli Znajomi i złączymy ją dwa razy z tabelą Osoby. Pierwszy raz (o1) będzie to osoba, której znajomych szukamy. Drugi raz (o2), to ów poszukiwany znajomy, nieuznający znajomości z o1. Mamy tu do czynienia z tzw. **samozłączeniem tabel**. Bez aliasów nie udałoby się tego zapisać, gdyż musimy w jakiś sposób rozróżnić te dwie role osób.

```
SELECT DISTINCT o1.Id, o1.Imię, o1.Nazwisko
FROM Osoby o1, Osoby o2, Znajomi z
WHERE o1.Id = z.IdOsoby
    AND o2.Id = z.IdZnajomego
    AND 2012 BETWEEN z.Początek AND z.Koniec
    AND NOT EXISTS (SELECT 1
                    FROM Znajomi ze
                    WHERE ze.IdOsoby = o2.Id
                        AND ze.IdZnajomego = o1.Id
                       AND 2012 BETWEEN ze.Początek AND ze.Koniec);
```

Zapytanie to polega na wyznaczeniu znajomości z osoby o2 przez osobę o1, a następnie stwierdzeniu, że nie istnieje znajomość ze osoby o1 z osobą o2. Ponownie podkreślmy, że jest to tylko opis tego, co zapytanie ma wypisać. Jak poradzi sobie z tym system zarządzania bazą danych, to już całkiem inna historia. Wcale nie musi on wykonywać czynności w taki sposób, który jest dla człowieka najwygodniejszy do opisu semantyki zapytania!

Kolejne zapytanie służy często w portalach społecznościowych do sugerowania nowych znajomości. Znajduje ono pary osób, które same nie są znajomymi, ale mają wspólnego znajomego. To zapytane jest podobne do poprzedniego. Tym razem jednak łączymy aż pięć egzemplarzy tabel, trzy razy tabelę Osoba i dwa razy tabelę Znajomi. Chcemy, by istniała znajomość z12 osoby o1 z osobą o2 oraz znajomość z23 osoby o2 z osobą o3, ale zabronione jest istnienie znajomości z13 między osobami o1 i o3:

```
SELECT DISTINCT o1.Id, o1.Imię, o1.Nazwisko,
                  o3.Id, o3.Imię, o3.Nazwisko
FROM Osoby o1, Osoby o2, Osoby o3, Znajomi z12, Znajomi z23
WHERE o1.Id = z12.IdOsoby
    AND o2.Id = z12.IdZnajomego
```

```
    AND o2.Id = z23.IdOsoby
    AND o3.Id = z23.IdZnajomego
    AND o1.Id!= o3.Id
    AND NOT EXISTS (SELECT 1
                    FROM Znajomi z13
                    WHERE z13.IdOsoby = o1.Id
                        AND z13.IdZnajomego = o3.Id);
```

Tym razem naiwne obliczenie wyniku tego zapytania na pewno nie wchodzi w rachubę. Polegałoby ono na wyznaczeniu wszystkich piątek, a następnie wyselekcjonowaniu tych, które są złożone z pasujących do siebie elementów. To oznaczałoby ogromną złożoność $O(o^3z^2)$, gdzie o to liczba wierszy w tabeli `Osoba`, a z to liczba wierszy w tabeli `Znajomi`. System zarządzania bazą danych musi takie zapytania wykonywać znacznie szybciej.

Ostatnie z przykładowych zapytań jest ilustracją możliwości języka SQL, która pojawiła się dopiero w roku 1999, czyli 13 lat po ukazaniu się pierwszej wersji standardu tego języka. Chodzi tutaj o możliwość rekurencyjnego wyszukiwania. Wynikiem poniższego zapytania ma być krąg przyjaciół osoby o identyfikatorze 1, ich przyjaciół, ich przyjaciół przyjaciół itd., aż do szóstego poziomu.

```
WITH RECURSIVE Krąg (Id, Imię, Nazwisko, Poziom) AS
(
    SELECT o.Id, o.Imię, o.Nazwisko, 0
    FROM Osoby o
    WHERE o.Id = 1
UNION ALL
    SELECT o.Id, o.Imię, o.Nazwisko, k.Poziom + 1
    FROM Krąg k, Osoby o, Znajomi zko
    WHERE k.Id = zko.IdOsoby
        AND o.Id = zko.IdZnajomego
        AND 'przyjaciel' = zko.Sposob
        AND k.Poziom < 6
)
SELECT *
FROM Krag;
```

W tym zapytaniu jest budowana pomocnicza relacja `Krąg`, która w końcowym kroku zapytania jest po prostu wypisywana jako wynik tego zapytania (ostatnia

w zapytaniu klauzula SELECT). Na początku relacja Krąg jest zbiorem pustym. Pierwsze podzapytanie SELECT znajduje osobę o identyfikatorze 1 zaznaczając, że jej poziom to 0 i dodaje ją do relacji Krąg. Drugie podzapytanie SELECT, iteracyjnie dodaje nowych przyjaciół do już istniejącego kręgu. Korzysta się bowiem z już znalezionych wierszy tabeli Krąg i łączy istniejących członków kręgu k znajomością zko z osobą o. Nowo dodana osoba ma poziom o jeden większy niż jej poprzednik. Obliczenia te są prowadzone aż do wyczerpania wszystkich możliwości, to jest do czasu, aż to drugie podzapytanie SELECT zwróci pusty wynik. Kiedyś musi to nastąpić, ponieważ każde obliczenie rekurencyjne zwiększa poziom o jeden, a jest ograniczenie, że poziom krotki k musi być mniejszy niż 6.

Możliwości takich rekurencyjnych zapytań są niestety bardzo ograniczone. Nie możemy bowiem sprawdzić, czy nowy element relacji rekurencyjnej nie został już wcześniej wygenerowany. Bardzo łatwo jest więc napisać zapytanie rekurencyjne, które się zapętli. Aby tego uniknąć, można ewentualnie dodać kolumnę licznikową, by ograniczyć liczbę wywołań rekurencyjnych (jak w powyższym przykładzie).

4. Transakcje

Infocolligens dysponuje zatem wygodnym językiem do pobierania danych z bazy, choć jak wynika z ostatniego przykładu, język SQL ma istotne ograniczenia. Baza danych musi także zapewniać bezpieczeństwo danych na wypadek awarii lub dostępu do danych przez wielu użytkowników bazy równocześnie. Dwóch użytkowników modyfikujących te same dane w tym samym czasie może doprowadzić do ich zniszczenia. W praktyce takie sytuacje traktuje się jako szczególny rodzaj awarii.

Aby radzić sobie z awariami systemów zarządzania bazami danych, *Infocolligens* posługuje się pojęciem **transakcji**, czyli zestawu operacji wykonywanych na bazie danych o pewnych szczególnych właściwościach, opisywanych angielskim skrótem **ACID** (ang. *Atomicity-Consistency-Isolation-Durability*). **Atomowość** (**A**) oznacza, że transakcja jest wykonywana jako całość, nie może być zrealizowana częściowo: albo kończy się sukcesem (zatwierdzeniem) albo porażką (wycofaniem). Wycofanie oznacza, że transakcja w ogóle nie miała miejsca i w danych nie pozostaje po niej żaden ślad. **Spójność** (**C**) to wymaganie, aby transakcja, która zastanie bazę danych w stanie spójnym, po swoim zakończeniu również pozostawiła spójny stan danych. **Spójna baza danych** zawiera wyłącznie dane poprawne, tj. zgodne z rzeczywistością. Dozwolone są stany niespójne, ale nie mogą takie pozostać po zatwierdzeniu transakcji. **Izolacja** (**I**) to zbudowana na użytek jednej transakcji iluzja, że ta transakcja ma całą bazę danych wyłącznie do swojej dyspozycji i jest całkowicie odizolowana od pozo-

stałych transakcji, jej działanie nie zależy więc od nich. Najwyższą formą izolacji jest **szeregowalność**. System zarządzania bazą danych może dopuszczać pewne przeploty operacji rożnych transakcji, ale tylko takie, których wynik końcowy będzie taki sam jak przy pewnym szeregowym (niewspółbieżnym) wykonaniu tych transakcji. Ten porządek szeregowy ma istnieć, ale nie jest określone, jaki ma być. Na koniec **trwałość** (**D**) oznacza, że dane wpisane przez zatwierdzoną transakcję mają znaleźć się w bazie nawet po awarii SZBD, nośnika danych itd. Jeśli więc transakcja zakończy się sukcesem, nic nie może stać się jej wynikom.

System zarządzania bazą danych realizuje atomowość i trwałość prowadząc dzienniki. **Dziennik powtórzeń** zawiera zapis wszystkich operacji modyfikujących dane i jest zrzucany fizycznie na dysk po każdym zatwierdzeniu transakcji. Dla bezpieczeństwa może być też replikowany. Gdy SZBD jest uruchamiany, sprawdza, czy został czysto zamknięty. Jeśli wykryje, że jego działanie zostało przerwane awarią, przystępuje do **odtwarzania**, tj. ponawia operacje zapisane w dzienniku od ostatniego czystego zamknięcia bazy, a ściślej od tzw. **punktu kontrolnego**. **Dziennik wycofań** zawiera przepisy na wycofanie poszczególnych operacji i jest wykorzystywany do wycofania operacji nieudanych transakcji. Gdy transakcja się wycofuje (po awarii i wznowieniu dotyczy to wszystkich aktywnych transakcji), dziennik wycofań jest czytany od tyłu i stosowane są zapisane w nim operacje anulujące. Izolację realizuje się poprzez stosowanie blokad/zamków. Gdy transakcja działa na jakiejś danej, zakłada na nią zamek, który wyklucza korzystanie z tych samych danych przez inną transakcję. To może powodować **zakleszczenia**, gdy dwie transakcje trzymają zamki na dwóch danych i obie czekają na daną akurat zamkniętą przez tę drugą transakcję. Wymaganie spójności implementuje się poprzez kontrolę **więzów integralności**, tj. warunków poprawności danych, najpóźniej przy zatwierdzeniu. Wymaganie unikatowości klucza głównego, poprawności klucza obcego, czy nieujemność wieku osoby to przykłady takich więzów.

Widać więc, że zabawki *Homo informaticus colligens* są bardzo złożone. Ich implementacja wymaga wiedzy z każdej dziedziny praktycznej informatyki: budowy sprzętu, systemów operacyjnych, programowania współbieżnego, kompilatorów (zapytania trzeba analizować składniowo i wykonywać), struktur danych, logiki, algorytmiki i sztucznej inteligencji (przy przetwarzaniu zapytań; por. punkt 5).

5. Przetwarzanie zapytań, optymalizator i adaptacyjność

Dla każdego *Homo informaticus*, a więc także dla *Infocolligensa* zdrowe lenistwo infrastruktury komputerowej jest jedną z najważniejszych wartości. Wcale nie chodzi o to, aby komputer się napracował, ale żeby dostarczył poprawny

wynik obliczeń. Im mniej kroków wykona i zasobów zużyje, tym lepiej. W bazie danych mamy jednak do czynienia z inną sytuacją niż w przypadku tradycyjnego programowania. Podstawowy język baz danych, SQL, określa bowiem to, *co* ma być zwrócone, a nie, *jak* to obliczyć. Dla jednego zapytania może istnieć bardzo wiele algorytmów wyliczających poprawnie jego wynik. Przetwarzanie zapytania zaczyna się więc od wyznaczenia planu (algorytmu). Następnie ten plan jest realizowany. Zanim jednak do tego dojdzie, system zarządzania bazą danych robi wszystko, aby jak najmniej się napracować. SZBD w swoich strukturach danych zapamiętuje plany pewnej liczby wykonanych uprzednio zapytań, a także wyniki niektórych z nich. Wybór wyników do zapamiętania odbywa się heurystycznie w ramach dostępnych zasobów (głównie pamięci operacyjnej). Tu po raz pierwszy spotykamy się ze sztuczną inteligencją SZBD.

5.1. Przed poszukiwaniem planu wykonania

Zanim jednak SZBD przystąpi do wyboru planu weryfikuje, czy w ogóle musi to robić. Na początku sprawdza więc, czy otrzymane zapytanie nie zostało już wcześniej zapamiętane. Jeśli ma jego aktualny wynik, to po prostu go odsyła. Jeśli ma jego wcześniej wyznaczony plan, przystępuje do wykonania tego planu. Tożsamość zapytań jest sprawdzana literalnie, co do znaku. W praktyce oblicza się wartość pewnej funkcji haszującej z tekstu zapytania. Programista aplikacyjny nie powinien więc wpisywać wartości stałych do zapytania.

Pierwsze przykładowe zapytanie z tego rozdziału jest zatem niewłaściwe:

```
SELECT * FROM Osoby WHERE Wiek = 14;
```

Programista powinien użyć tzw. **zmiennej wiązania** (parametru poprzedzanego dwukropkiem) i przy wywołaniu tego zapytania podać wartość `:W`:

```
SELECT * FROM Osoby WHERE Wiek = :W;
```

Osiągamy dzięki temu dwa cele. Po pierwsze SZBD może to zapytanie rozpoznać jako tożsame, gdy zostanie ono wysłane wielokrotnie, i unikać w ten sposób kosztownego szukania planu. Po drugie uniemożliwiamy w ten sposób ataki **wstrzykiwania SQL** (ang. *SQL injection*), które polegają na wpisywaniu do formularza WWW fragmentów zapytań z użyciem znaków specjalnych, głównie apostrofów i odwrotnych ukośników. W wyniku takiego ataku znaczenie zapytań przekazywanych do bazy danych z aplikacji WWW może ulec modyfikacji, co z kolei może prowadzić do uszkodzenia danych lub niepożądanego ujawnienia tajemnic.

Niestety, nauczenie programistów aplikacyjnych tak prostej rzeczy, jak nieużywanie stałych w zapytaniach, okazuje się być zadaniem niewykonalnym. Z tego powodu w SZBD wprowadzano pewne udogodnienie. Po włączeniu odpowiedniej opcji konfiguracyjnej, SZBD analizuje wstępnie wszystkie przychodzące zapytania i zamienia w nich stałe na zmienne wiązania. To oczywiście nie rozwiązuje problemu ataku wstrzykiwanym SQL, ale za to zwiększa ponowne użycie raz wyznaczonych planów. Nie odbywa się to jednak bez kosztów. Po pierwsze SZBD wykonuje dodatkowe czynności przy przetwarzaniu zapytania, co powoduje zużycie zasobów. Po drugie i znacznie ważniejsze, tracimy w ten sposób pewną możliwość. Może się bowiem zdarzyć, że dla pewnych wartości parametrów chcemy planu wykonania innego niż dla pozostałych. Taka sytuacja ma miejsce, gdy dane w jakiejś kolumnie są rozłożone nierównomiernie (np. w bazie danych polskich żołnierzy pole płeć będzie zawierało zdecydowanie więcej liter M niż K). Bez włączenia opcji usuwania stałych, po prostu użyjemy parametru dla wartości standardowych, a stałej dla wartości szczególnych. Po włączeniu tej opcji, cały ten wysiłek jest jałowy. SZBD wszystkie wartości potraktuje tak samo.

5.2. Algebra relacji

Zajmijmy się teraz sytuacją, w której SZBD nie ma gotowego planu i musi go utworzyć. Proces ten rozpoczyna się od analizy składniowej zapytania, która prowadzi do wytworzenia wstępnego planu logicznego. Taki plan to drzewo wyrażenia **algebry relacji**, która stanowi podstawę teoretyczną przetwarzania zapytań w bazach danych. Nośnikiem tej algebry są skończone relacje, a działaniami są następujące operatory. W nawiasach podano ich tradycyjnie używane symbole.

Selekcja(σ) to wybór pewnego podzbioru wierszy z zadanej tabeli na podstawie warunku logicznego.
Rzutowanie(π) to wybór pewnego podzbioru kolumn z zadanej tabeli.
Zmiana nazw(ρ) kolumn w zadanej tabeli.
Iloczyn kartezjański($\times$) dwóch tabel to zbiór wszystkich możliwych par ich wierszy.
Złączenie($\bowtie$) dwóch tabel to zbiór wszystkich pasujących do siebie par ich wierszy. Dopasowanie wierszy jest zadane formułą logiczną.
Suma($\cup$) mnogościowa dwóch tabel.
Różnica($-$) mnogościowa dwóch tabel.
Przecięcie($\cap$) mnogościowe dwóch tabel.

Ten zestaw operatorów jest nadmiarowy, ponieważ przecięcie jest szczególnym przypadkiem złączenia. Iloczyn kartezjański to złączenie z warunkiem zawsze prawdziwym. Z kolei złączenie to złożenie iloczynu kartezjańskiego, zmiany nazw i selekcji. Można by więc ten zbiór operatorów ograniczyć do sze-

ściu. Język SQL wymaga też dodatkowych operacji niedefiniowalnych w algebrze relacji (bo relacje są zbiorami), jak sortowanie, grupowanie i agregacja.

Przypomnijmy jedno z pierwszych zapytań z punktu 3.

```
SELECT o.Imię, o.Nazwisko, h.Nazwa
FROM Osoby o, Hobby h
WHERE o.Id = h.IdOsoby
    AND 2012 BETWEEN h.Początek AND h.Koniec;
```

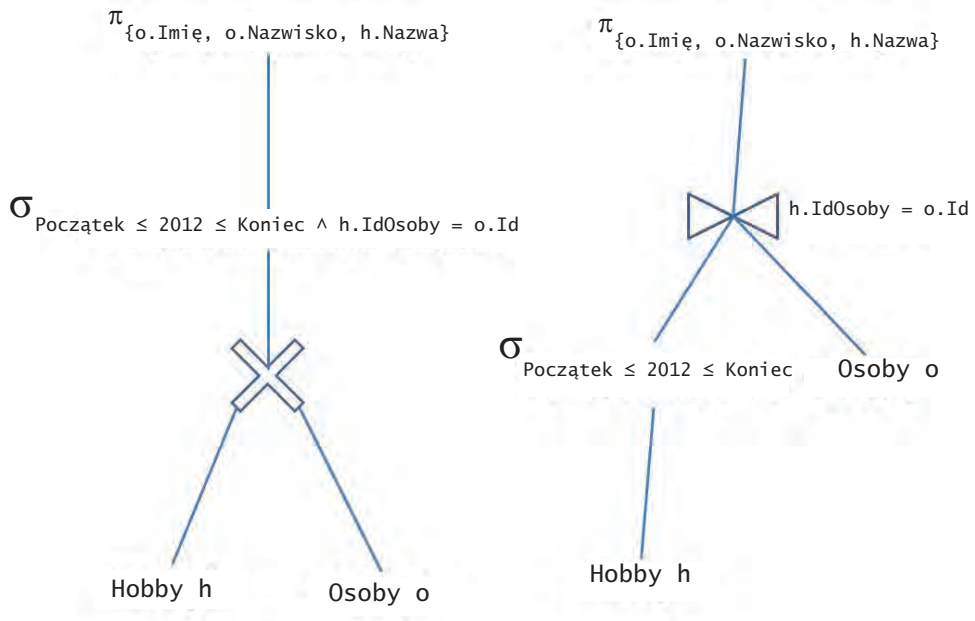
Początkowy plan logiczny realizacji tego zapytania, wynikający li tylko z jego analizy składniowej, znajduje się w lewej części rysunku 2. Zauważmy, że jest on nieefektywny. Przewiduje bowiem produkt kartezjański tabel wejściowych wygenerowany na podstawie klauzuli `FROM`, następnie ma zostać wykonana selekcja (wynik `WHERE`), a na końcu rzutowanie (wynik `SELECT`).

5.3. Optymalizatory

Teraz do dzieła przystępuje **optymalizator regułowy**, który stara się tak przekształcić plan logiczny zapytania, by poprawić jego wydajność. Stosowane reguły muszą być oczywiście poprawne (zachowywać ten sam wynik zapytania). Muszą także zwiększać wydajność w każdych warunkach. Niestety takich reguł jest niewiele. Jedną z nich jest reguła, której efekt jest widoczny w prawej części rysunku 2. Reguła ta polega na rozpoznaniu warunków selekcji jako warunków złączenia i zamianie iloczynów kartezjańskich na złączenia. Po jej zastosowaniu w przypadku powyższego zapytania pozbywamy się iloczynu kartezjańskiego i otrzymujemy szansę na wygenerowanie planu o złożoności mniejszej niż kwadratowa.

Optymalizator regułowy ma niestety niewielkie pole do popisu, choć wyniki jego działania są bardzo ważne. Po nim pałeczkę przejmuje **optymalizator kosztowy**, który znając statystyki dotyczące wielkości tabel, ich przestrzeni składowania, rozkładów wartości w kolumnach itd. dokonuje wyboru najlepszego planu zapytania. Zanim zajmiemy się nim, przyjrzyjmy się jego arsenałowi. Metod wykonywania zapytań jest oczywiście bardzo wiele i nie ma miejsca, żeby je wszystkie omówić. Ograniczymy się do niezwykle ważnych złączeń równościowych. Dlaczego tak ważnych? W relacyjnej bazie danych zależności między rekordami zapisujemy jako wartości kluczy obcych. Odnalezienie połączonego rekordu wymaga więc złączenia równościowego, które jest najlepiej rozpracowanym przez naukowców operatorem algebry relacji i też jako pierwszy został porządnie zaimplementowany przez firmę IBM w pierwszym systemie SZBD o nazwie R.

Rozważania będą ilustrowane zapytaniem, którego plan logiczny jest umieszczony po prawej stronie na rysunku 2. Przedstawimy podstawowe algorytmy wykonania złączenia równościowego z tego zapytania.



Rysunek 2. Początkowy i poprawiony plan logiczny przykładowego zapytania

5.4. Algorytmy realizacji złączenia równościowego

Pierwszy algorytm jest najprostszy; nosi nazwę **zagnieżdżonych pętli** (ang. *nested loops join*, NLJ). Polega on na wczytywaniu kolejnych fragmentów pierwszej tabeli do pamięci operacyjnej i przeglądaniu kolejnych stron drugiej tabeli w celu znalezienia pasujących par. Owszem, asymptotycznie ten algorytm ma złożoność kwadratową, jednak w przypadku baz danych bierzemy pod uwagę najkosztowniejsze operacje, czyli transfery między dyskiem i pamięcią, za to zaniedbujemy operacje w pamięci operacyjnej. Jeśli dostępna pamięć ma wielkość M, a liczby stron złączanych tabel to odpowiednio H i O, to algorytm zagnieżdżonych pętli wczyta OM fragmentów tabeli `Osoby` i dla każdego fragmentu odczyta całą zawartość `Hobby`. To daje nam HOM odczytów z dysku. Zauważmy po pierwsze, że jeśli jedna z tabel w całości mieści się w pamięci, to druga będzie odczytana tylko raz. Po drugie mamy tu do czynienia z odczytem dużych obszarów dysku, które są bardzo szybkie w porównaniu ze swobodnym czytaniem małych fragmentów. Te elementy mogą przesądzić o tym, że ten algorytm jest najlepszy dla konkretnej bazy danych.

Drugi algorytm, **złączenie indeksowe** (ang. *index nested loops*, INLJ) jest udoskonaleniem pierwszego, ale jest możliwy do wykonania tylko wtedy, gdy na kolumnie złączenia w jednej z tabel założono *indeks* umożliwiający wyszukiwanie wierszy o zadanym kluczu w czasie logarytmicznym. Indeks taki ma zwykle strukturę B^+drzewa, tj. zrównoważonego drzewa wyszukiwań, będą-

cego dalekim kuzynem drzew wyszukiwań binarnych (BST). Przypuśćmy, że mamy taki indeks na kolumnie `Id` tabeli `Osoby`[8]. Wtedy wystarczy przeglądać kolejno wiersze tabeli `Hobby` i dla każdego z nich wyszukać w indeksie na `Osoby.Id` wszystkie wiersze spełniające warunek złączenia. „Jaki dobry algorytm!", zachwyci się ktoś. Mamy przecież złożoność liniowo-logarytmiczną $O(H \log O)$. Uwaga! Wcale nie jest tak dobrze. Tym razem bowiem nasze dostępy do dysku są swobodne: za każdym razem pobieramy jeden blok z przypadkowego miejsca na dysku. Dostępy swobodne są jednak znacznie kosztowniejsze od ciągłych. To powoduje, że ten algorytm nie musi być lepszy od algorytmu NLJ.

Dwa pozostałe algorytmy: **złączenie przez scalanie** (ang. *sort merge join*, SMJ) i **złączenie haszowane** (ang. *hash join,* HJ) polegają na wstępnym przetworzeniu plików z tabelami. Algorytm SMJ najpierw sortuje obie tabele po wartości kolumny złączenia, a następnie przebiega obie posortowane kolumny poszukując pasujących par zupełnie tak samo, jak w fazie scalania algorytmu mergesort. Wskaźnik bieżącego rekordu porusza się tylko w jednym kierunku, a więc faza scalania ma koszt liniowy. Koszt złączenia przez scalanie to zatem $O(O \log O + H \log H)$.

Z kolei algorytm HJ czyta obie tabele i za pomocą funkcji haszującej liczonej na wartości kolumny złączenia rozrzuca ich wiersze do kubełków tablicy haszującej. Następnie w drugiej fazie łączy ze sobą zawartości kubełków. Jeśli funkcja haszująca jest dobra i nie tworzy zbyt wielkich kubełków, tj. niemieszczących się w pamięci, to druga faza odbywa się przy tylko jednym odczycie zawartości kubełków. Mamy więc złożoność $O(O + H)$. Ten algorytm ma jeszcze jedną zaletę. Jeśli jedna z tabel jest mała i jej tablica haszująca mieści się w pamięci, to haszujemy ją w pamięci, a następnie czytamy tabelę większą poszukując pasujących par za pomocą znajdującej się w RAM tablicy haszującej.

Dodatkową zaletą obu ostatnich algorytmów jest wykonywanie przez nie wyłącznie ciągłych, a więc bardzo szybkich, odczytów z dysku.

5.5. Optymalizator kosztowy

Ta mała próbka dostępnych algorytmów wykonywania złączenia równościowego pokazuje możliwości, jakimi dysponuje optymalizator kosztowy. Na marginesie zauważmy, jak ważna jest w tym procesie algebra relacji. Umożliwia bowiem podział zapytania na mniejsze fragmenty i korzystanie z pewnej rozsądnej liczby algorytmów dla tych fragmentów. To jednak nie wszystko, co można zrobić przy optymalizacji kosztowej. Optymalizator może wpływać także na kształt drzewa planu logicznego. Złączenie ma dwie ważne właściwości: symetryczność i łączność (jak dodawanie w arytmetyce):

[8] W istocie taki indeks zawsze będzie istniał. System SZBD założy go automatycznie, aby usprawnić kontrolę więzów klucza głównego na tej kolumnie.

$$R \bowtie S = S \bowtie R$$
$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Kolejność wykonywania złączeń jest więc zupełnie obojętna. To dobra wiadomość, daje bowiem optymalizatorowi dużą przestrzeń możliwych planów wykonawczych. Jest też niestety poniekąd zła: przestrzeń ta jest tak ogromna, że nie da się jej gruntownie przeszukać. Logiczny plan zapytania ma postać drzewa. Jeśli uwzględnimy w nim tylko złączenia, to będzie to pełne drzewo binarne[9]. Przykłady takich drzew są pokazane na rysunku 3. Jeśli w zapytaniu złączamy *n* tabel, to samych kształtów drzew binarnych o *n* liściach jest bardzo dużo. Ich liczba jest równa liczbie Catalana:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

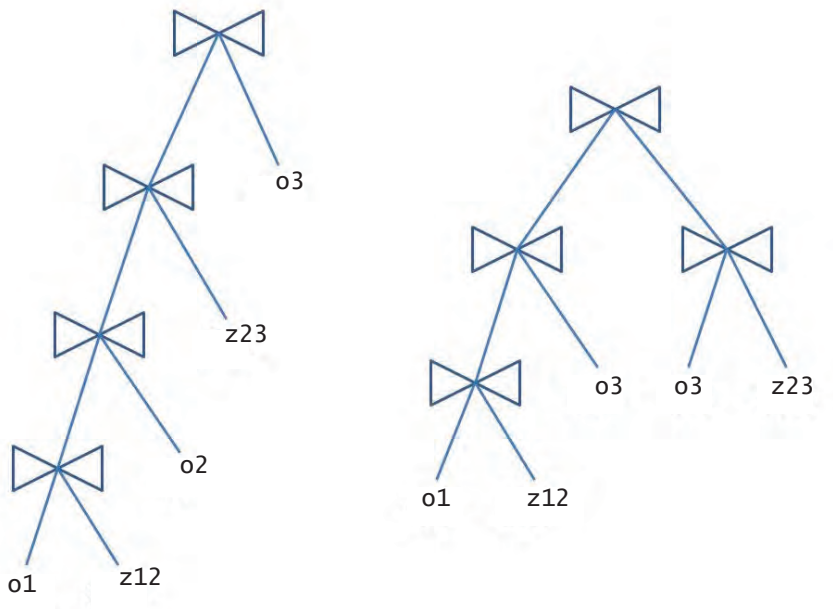
Nas interesuje jednak nie tylko kształt drzewa, ale i rozmieszczenie relacji w jego liściach. Trzeba zatem pomnożyć *Cn* przez *n*!, by poznać liczbę wszystkich możliwych drzew kolejności złączeń. To ogromna liczba. Można wykazać, że problem optymalizacji zapytań rozumiany jako wybór najbardziej efektywnego drzewa kolejności złączeń jest NP-trudny[10]. Optymalizator kosztowy musi sobie więc radzić inaczej. Należy pamiętać także o tym, że wyborowi podlega nie tylko kolejność wykonywania złączeń, ale także konkretne algorytmy ich wykonania, takie jak wspomniane już NLJ, INLJ, SNJ i HJ.

Pierwszy optymalizator kosztowy zrealizowany we wspomnianym już systemie R firmy IBM brał pod uwagę tylko jeden kształt drzew: drzewa skierowane w lewo, takie jak drzewo z lewej części rysunku 3. Ograniczyło to wielkość przestrzeni poszukiwań do zaledwie, bagatela, *n*!. Optymalizator systemu R był zaprogramowany dynamicznie na maskach bitowych. Dla każdego podzbioru już złączanych tabel i dla każdej z pozostałych tabel sprawdzał, czy dołączenie jednej z pozostałych tabel do tego zestawu da lepszy wynik niż do tej pory znaleziony. Ten algorytm ma złożoność wykładniczą $O(n2^n)$. Jest więc sensowny dla małych liczb złączanych tabel. W systemie PostgreSQL[11] jest domyślnie stosowany dla $n \leq 12$.

[9] Każdy wierzchołek **pełnego drzewa binarnego** jest liściem albo ma dokładnie dwóch synów.

[10] O problemach NP-trudnych jest mowa w artykule *Czy wszystko można obliczyć. Łagodne wprowadzenie do złożoności obliczeniowej.*

[11] System PostreSQL ma otwarty kod źródłowy, można więc podpatrzeć, co robi jego optymalizator zapytań. W przypadku produktów komercyjnych algorytmy optymalizacji są pilnie strzeżonymi tajemnicami.



Rysunek 3. Przykładowe kształty drzew planu zapytania

Gdy liczba złączanych tabel jest istotnie większa, algorytm przeszukujący całą przestrzeń planów jest zbyt powolny. Nie możemy przecież dopuścić, by optymalizacja zapytania trwała dłużej niż wykonanie pierwszego z brzegu planu, choćby najgorszego. Z tego powodu, dla większych zapytań stosuje się heurystyczne metody optymalizacji opracowane w dziedzinie sztucznej inteligencji, takie jak programowanie genetyczne czy symulowane wyżarzanie. Ze względu na to, że i tak nie przeszukujemy całej przestrzeni, a drzewa inne niż skierowane w lewo mogą być znacznie lepsze, heurystyczne optymalizatory kosztowe biorą pod uwagę także tzw. **drzewa krzaczaste** (ang. *bushy trees*). Przykład takiego drzewa widać po prawej stronie na rysunku 3. Drzewa z tego rysunku są planami wykonawczymi uproszczonego zapytania pokazanego już w punkcie 3:

```
SELECT DISTINCT o1.Id, o1.Imię, o1.Nazwisko,
                        o3.Id, o3.Imię, o3.Nazwisko
FROM Osoby o1, Osoby o2, Osoby o3, Znajomi z12, Znajomi z23
WHERE o1.Id = z12.IdOsoby
    AND o2.Id = z12.IdZnajomego
    AND o2.Id = z23.IdOsoby
    AND o3.Id = z23.IdZnajomego
    AND o1.Id!= o3.Id;
```

To jednak nie koniec możliwości optymalizacji w bazach danych. *Homo informaticus,* podobnie jak inni *Homo sapiens*, w nocy śpi. Baza danych nie wymaga jednak snu, więc gdy jej użytkownicy śpią, może wykonywać czynności na ich rzecz. Jedną z takich czynności jest optymalizacja *off-line*. W trakcie dnia SZBD może kolekcjonować ważne zapytania, tzn. często wykonywane i/lub pożerające dużo zasobów. Gdy były one wykonywane, były optymalizowane *on-line*. Siłą rzeczy musiała to być optymalizacja szybka, bo przy terminalu na wynik czekał niecierpliwy użytkownik. Optymalizator kosztowy nie mógł więc zbyt gruntownie przejrzeć przestrzeni poszukiwań najlepszego planu.

Gdy jednak zapada noc[12] i po serwerowni zaczynają snuć się wampiry, złe duchy i białe damy, SZBD przystępuje do wykonywania tzw. **zadań wsadowych** (np. generowanie faktur i tworzenie rozbudowanych raportów) oraz zadań administracyjnych, z optymalizacją *off-line* włącznie. Taka optymalizacja jest dokładniejsza, przeszukuje bowiem większy obszar przestrzeni planów wykonania. Rano, gdy użytkownicy przyjdą do pracy, mogą ze zdziwieniem zauważyć, że ich zapytania działają teraz szybciej. Oczywiście, aby ten mechanizm mógł zadziałać, zapytania muszą być rozpoznane przez SZBD jako powtarzalne. To jest możliwe tylko wtedy, gdy programiści będą stosowali zmienne wiązania, a nie będą umieszczali stałych w tekście zapytania – piszemy o tym na początku tego punktu.

Jak widać kwestia doboru planów jest bardzo dokładnie zbadana i dopracowana w systemach zarządzania bazami danych. Wiemy jednak, że wybrany plan wykonania nie musi być optymalny, bo do jego wyboru są stosowane algorytmy przybliżone. Może się więc zdarzyć, że wybrany plan działa bardzo nieefektywnie. Dotychczas omówiona architektura SZBD przewidywała działanie w stylu wybór planu – wykonanie. *Infocolligens* to jednak *Homo sapiens*, więc wziął też pod uwagę możliwość modyfikacji planu w trakcie wykonywania. To doprowadziło do koncepcji **adaptacyjnego przetwarzania zapytań** (ang. *adaptive query processing*)[13], czyli przetwarzania, które dostosowuje się do specyfiki otrzymanych danych.

Istnieją dwa podejścia do adaptacji. Pierwsze jest bardziej oczywiste i polega wykrywaniu wadliwości wykonywanego planu i jego korekcie po wykryciu usterek. Ma ono jednak istotną wadę – zwykle wyniki otrzymane przed korektą są tracone. Znacznie bardziej obiecujące jest drugie podejście, które można nazwać **bezplaniem probabilistycznym**[14]. Bezplanie oznacza brak planu wyko-

[12] Oczywiście mamy na myśli noc wirtualną, tzn. czas mniejszego obciążenia serwerów w związku z zakończeniem dnia roboczego w gospodarce realnej.

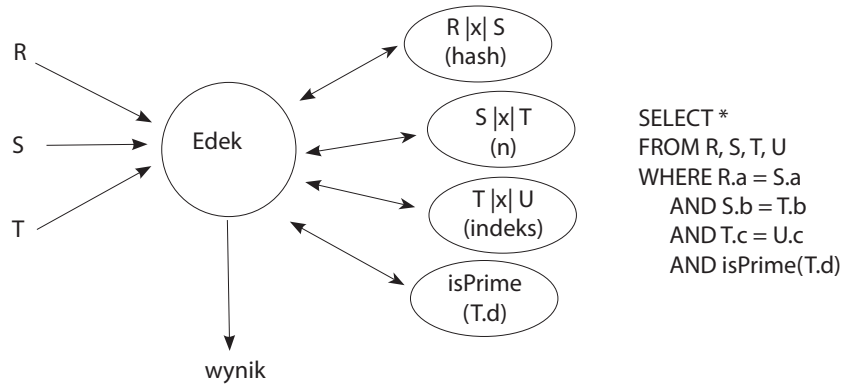
[13] Znakomitym źródłem informacji na temat adaptacyjnego przetwarzania zapytań jest przeglądowy artykuł [1].

[14] Ten termin jest całkowicie inwencją autora, który z góry przeprasza co wrażliwszych czytelników za ewentualny ból zębów w trakcie jego odczytywania. Nie ma on odpowiednika po angielsku, za taki można ewentualnie przyjąć *probabilistic no-plan*.

nania. Probabilistyczne jest dlatego, że kolejne kroki do wykonania są wybierane w wyniku losowania. Prawdopodobieństwa w tym losowaniu dobiera się na podstawie dotychczasowej historii realizacji planu wykonania. Przykładem mechanizmu adaptacyjnego jest metoda oparta na module Edek, zilustrowana na rysunku 4. Na wejściu ten moduł otrzymuje strumień danych wejściowych z relacji *R*, *S* i *T*. Gdy na wejściu pojawiają się krotki tych relacji, to są kierowane do odpowiednich operatorów złączeń i kosztownej selekcji (sprawdzenie, czy kolumna *d* w tabeli *T* jest liczbą pierwszą). Kierowanie odbywa się losowo na podstawie **tablicy trasującej** (ang. *routing table*). Każdy z możliwych celów jest opatrzony prawdopodobieństwem, z jakim krotka ma tam trafić. Gdy krotka wpada do węzła operatora, jest przetwarzana i może zostać odrzucona (jeśli nie spełniła warunku selekcji albo nie złączyła się z żadną z krotek relacji przeciwnej) lub zostać zwrócona do modułu Edek, być może w kilku kopiach (gdy złączona została więcej niż jedną krotką). Gdy krotka odwiedzi wszystkie operatory (kontroluje to towarzysząca jej maska bitowa), zostaje przekazana do strumienia wynikowego.

Gdzie tutaj jest adaptacja? Moduł Edek kontroluje parametry wykonania poszczególnych operatorów, takie jak czas przetworzenia jednej krotki oraz stopień wyjścia, czyli ile krotek wróciło do modułu Edek po wrzuceniu jednej krotki. Na podstawie tych parametrów moduł Edek może zmieniać prawdopodobieństwa trasowania w tablicy. Czasem jakaś ścieżka może mieć prawdopodobieństwo zerowe, jeśli jest całkowicie nieopłacalna. Plan wykonania zapytania realizowany przez moduł Edek jest więc zapisany w prawdopodobieństwach trasowania. Modyfikowanie planu (adaptacja) odbywa się poprzez zmianę tych prawdopodobieństw w trakcie działania. Nie powoduje to utraty wyników dotychczasowych obliczeń.

Adaptacyjne wykonywanie zapytań jest bardziej kosztowne niż klasyczne, jeśli statycznie wybrany plan zapytania jest dobry. Trzeba bowiem wówczas monitorować parametry wykonania i dostosowywać plan. Adaptacyjność jest obiecującą techniką w sytuacjach, w których optymalizator nie radził sobie z wyborem planu dla ogromnego zapytania, miał niedokładne statystyki o danych lub rzeczywiste dane zawierają niespodziewane korelacje, np. oszacowanie wielkości złączenia dwóch tabel było małe, ale w trakcie realizacji zapytania okazało się znacznie większe. Adaptacyjność to też sposób na przetwarzanie zapytań w **strumieniowych bazach danych** (ang. *stream databases*). W takich bazach dane są nieskończonym strumieniem, który nie jest w całości przechowywany. Przetwarzanie zapytań polega na przepuszczeniu tego strumienia przez operatory. W długich okresach charakterystyka strumieni danych zmienia się i mogą to być zmiany bardzo duże i jedynie adaptacyjny procesor zapytań ma szansę dobrze poradzić sobie w takich warunkach.



Tabele wejściowe	Odwiedzone	Następne cele
{R}	{}	(R |x| S, 1.0)
{ST}	{S |x| T, isPrime(T.d)}	(R |x| S, 0.3), (T |x| U, 0.7)
...	...	...

Rysunek 4. Edek. Jedna z metod adaptacyjnego przetwarzania zapytań

6. NOSQL

Bazy danych NOSQL są to narzędzia uzupełniające pewną lukę rynkową, wynikającą z ograniczeń klasycznych relacyjnych baz danych. Dzielimy je na trzy grupy: bazy klucz-wartość, bazy dokumentowe i bazy grafowe.

Bazy danych klucz-wartość są po prostu wielkimi skalowalnymi słownikami. Rejestrują bowiem wartości rekordów pod konkretnymi kluczami wyszukiwania. Pierwsza taka baza BerkeleyDB powstała zanim jeszcze pojawił się termin NOSQL. W bazie tej kluczem wyszukiwania jest ciąg bitów, a wartością... też ciąg bitów. Taka architektura bazy danych umożliwia nieograniczoną skalowalność, ponieważ dane można łatwo rozproszyć na bardzo wielu maszynach na podstawie wartości funkcji haszującej z klucza. Replikacja jest równie łatwa, o ile zrezygnujemy z aksjomatów ACID i zastąpimy je tzw. **spójnością ostateczną** (ang. *eventual consistency*). Gdy system zarządzania bazą danych implementuje taką spójność, to jest pewność, że po skończonym czasie od aktualizacji danych w jednym miejscu ostatecznie wszystkie kopie zostaną też poprawione. Zgodnie z konsekwencjami omówionego już twierdzenia CAP (podrozdział 2), tylko taka uboższa forma spójności jest możliwa. Bazy klucz-wartość eliminują więc ograniczenia klasycznych baz relacyjnych wynikające z rygorystycznego trzymania się aksjomatów ACID.

Obecnie jest bardzo wiele baz tego typu. Warto wspomnieć o bazie BigTable firmy Google, Apache Hbase czy Cassandrze, początkowo stosowanej w portalu Facebook. Przy pracach nad BigTable opracowano też nowy paradygmat przetwarzania rozproszonego, tzw. **MapReduce**. Przetwarzanie tą metodą jest bardzo popularne, ponieważ bardzo dobrze się skaluje i łatwo rozprasza na wiele maszyn. Metodę przetwarzania nazwiemy **skalowalną**, jeśli wzrost wielkości jej danych wejściowych powoduje proporcjonalny wzrost zużycia zasobów (czas działania, pamięć i liczba serwerów). Wyobraźmy sobie, że budujemy aplikację WWW. Będzie ona skalowalna, jeśli n-krotny wzrost liczby aktywnych użytkowników będzie wymagał dołączenia dodatkowo co najwyżej $n - 1$ nowych serwerów.

Pierwszy krok w MapReduce o nazwie **Map** polega na pobraniu wybranych rekordów z baz(y) danych i przekształceniu ich na pary klucz-wartość. Następnie w być może wielokrotnie wykonywanym kroku **Reduce,** pary o tym samym kluczu są grupowane i wyliczane są nowe wartości na podstawie pogrupowanych rekordów. Gdy dla każdego klucza istnieje już tylko jedna para, jest ona zwracana jako wynik. Operacja wykonywana podczas fazy redukcji musi być symetryczna i łączna żeby wynik obliczeń nie zależał od kolejności wykonania kroków Reduce.

Prześledźmy działanie metody MapReduce na przykładzie bazy danych, w której kluczem jest adres URL, a wartością zapamiętany dokument WWW spod tego adresu. Chcemy obliczyć, ile jest dokumentów w takiej bazie danych, w których występują wyrazy ze zbioru S. W fazie Map, dla każdego dokumentu i słowa $v \in S$ jest generowana para $\langle v, 1\rangle$, o ile v występuje w tym dokumencie. W fazie Reduce pary z tym samym słowem są sukcesywnie grupowane przez serwery wykonawcze, przy tym liczby wystąpień są sumowane. Po zakończeniu tego procesu ostatni serwer generuje jedną parę dla każdego słowa ze zbioru S. Drugi element tej pary to liczba dokumentów, w których to słowo występuje. Wykonywane w fazie Reduce dodawanie liczb jest symetryczne i łączne, a więc niezależnie od postaci kaskady serwerów redukujących, otrzymany wynik będzie zawsze poprawny. Zasady działania metody MapReduce są rzeczywiście proste, ale to tylko pozory. Efektywna implementacja tego mechanizmu wymaga wiele wysiłku ze względu na złożone problemy komunikacji i synchronizacji między uczestniczącymi maszynami (serwerami).

Bazy dokumentowe (ang. *document-oriented databases*) służą do przechowywania dokumentów w rozmaitych formatach: XML, JSON, BSON i tekst. Równie dobrze implementuje się na nich MapReduce. Bardzo wygodne jest też ich skalowanie i replikacja. Często też trudno jest odróżnić bazy dokumentowe i klucz-wartość. Przykładami najbardziej znanych baz dokumentowych są CouchDB, MongoDB, eXist i Redis. Baza MongoDB jest używana przez liczne

firmy na rynku mediów, takie jak Disney (do przechowywania stanów gier *on-line*), CNN, New York Times. Także znane programistom repozytorium kodów źródłowych GitHub używa MongoDB.

Jeśli szukać szczególnej cechy wyróżniającej takie bazy wśród innych baz typu NOSQL, to będzie to istnienie mechanizmów **wyszukiwania pełnotekstowego**, który polega na znajdowaniu dokumentów zawierających zadane słowa lub frazy. Taka funkcjonalność jest realizowana za pomocą **indeksów/list odwróconych** (ang. *inverted lists/indices*), które dla każdego słowa zawierają listę ich wystąpień w dokumentach. Podobne udogodnienie starają się też dostarczyć producenci klasycznych relacyjnych SZBD, jednak testy wykazują, że ich implementacje są istotnie wolniejsze od tych stosowanych w bazach typu NOSQL. Przyczyną jest m.in. wierność aksjomatom ACID.

Ostatnią grupą baz NOSQL, o których tutaj wspominamy, są **bazy grafowe** (ang. *graph databases*). Przykładowymi produktami z tej rodziny są Neo4j, GraphDB, OrientDB. Zapewne najbardziej znaną w Polsce firmą używającą jednej z tych baz (Neo4j) jest Deutsche Telecom, właściciel sieci T-Mobile. Także firma Cisco wykorzystuje bazę Neo4j.

Grafowe bazy danych przechowują i udostępniają strukturę grafową utworzoną przez rekordy bazy, traktowane jako wierzchołki grafu. Z kolei dla każdego rekordu (wierzchołka) jest przechowywana w takiej bazie lista jego sąsiadów. Zwykle rekordy mogą zawierać dowolną liczbę pól i w istocie są dokumentami. To powoduje, że rozróżnienie, czy dana baza NOSQL jest bazą grafową, klucz-wartość czy dokumentową nie jest takie proste. Ale bazy grafowe, tak jak inne bazy NOSQL, mają na celu zaoferowanie czegoś niedostępnego w relacyjnych SZBD. Tym czymś jest możliwość przejścia do sąsiada w czasie stałym. Przypomnijmy, że podobna operacja w bazie relacyjnej wymaga złączenia lub użycia indeksu i ma koszt logarytmiczny. Bazy grafowe są więc rozwiązaniem przystosowanym do przechowywania dużych grafów i wykonywania na nich obliczeń wymagających swobodnego poruszania się między wierzchołkami.

Zaraz, zaraz, stop! *Infocolligensie*, zatoczyłeś koło i wróciłeś do źródeł! Wymyśliłeś na nowo sieciowy model danych! I tak, i nie. Z pewnością model wydaje się podobny, ale są pewne istotne różnice. Po pierwsze wierzchołki w grafowej bazie danych mogą przechowywać dowolne informacje podczas, gdy baza sieciowa miała rygorystycznie przestrzegany schemat. Po drugie baza grafowa ma zaimplementowane mechanizmy replikacji i fragmentacji; dobrze więc skaluje się na wiele serwerów. Bazy sieciowe były tylko scentralizowane. I po trzecie – interfejs, głupcze! Bazy hierarchiczne oferują indeksy odwrócone i języki zapytań grafowych (np. Gremlin). Tego nie było w bazach sieciowych. Możliwości oferowane przez grafowe bazy danych są bardzo obiecujące z punktu widzenia budowniczego portalu społecznościowego. Sieciowa baza danych w ogóle się do tego nie nadaje.

Tak to już jest, że *Homo informaticus* musi pewne idee wykryć ponownie, by je udoskonalić i wprowadzić na nowy, wyższy poziom. Zaczęliśmy od baz hierarchicznych i sieciowych, potem pojawiły się bazy relacyjne, obiektowe, obiektowo-relacyjne, klucz-wartość, dokumentowe i grafowe. W bazie danych każdego z tych rodzajów *Homo informaticus colligens* przechowuje po prostu rekordy. Na szczęście za każdym razem obok marketingu pojawiały się też nowe właściwości i funkcjonalności. Rozwój w dziedzinie baz danych, mimo chwilowego przestoju na przełomie stuleci, jest ponownie niezwykle dynamiczny. Oby tak dalej...

Literatura

1. Deshpande A., Ives Z.G., Raman V., *Adaptive Query Processing*, „Foundations and Trends in Databases" 1(1) 2007, 1-140, http://www.nowpublishers.com/product.aspx?product=DBS&doi=1900000001
2. Garcia-Molina H., Ullman J.D., Widom J., *Systemy baz danych. Pełny wykład*, WNT Warszawa 2006
3. Stonebraker M., *Technical perspective – One size fits all: an idea whose time has come and gone*, „Comm. ACM", 51(12) 2008, 76
4. Subieta K., *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo PJWSTK, Warszawa 2004

Prof. nzw. dr hab. Krzysztof Stencel

jest profesorem nadzwyczajnym w Instytucie Informatyki Uniwersytetu Warszawskiego. Jego zainteresowania naukowe koncentrują się wokół baz danych oraz inżynierii oprogramowania.

Uczestniczył w zawodach programistycznych na początku ich rozwoju, reprezentując Polskę na I Międzynarodowej Olimpiadzie Informatycznej w 1989 roku. Od dłuższego czasu natomiast zajmuje się sędziowaniem i organizacją zawodów informatycznych. Przewodniczył jury Olimpiady Informatycznej od samego jej początku w roku 1993 aż do 2008 roku. Był też szefem jury zawodów międzynarodowych: Olimpiady Informatycznej Krajów Europy Środkowej (1997, 2004), Bałtyckiej Olimpiady Informatycznej, (2000, 2001, 2008) oraz Międzynarodowej Olimpiady Informatycznej (2005). W roku 2011 był przewodniczącym Olimpiady Informatycznej Krajów Europy Środkowej. W roku 2008 postanowił dla przyjemności rozwiązywać zadania programistyczne na słynnym serwerze Uniwersytetu Valladolid. Od 19 września 2011 roku do chwili pisania tego tekstu rozwiązał ponad 3000 zadań i jest liderem rankingu. Krzysztof Stencel jest autorem licznych publikacji oraz podręczników z baz danych, a także tłumaczem książek z tego zakresu.

stencel@mimuw.edu.pl