STUDIA INFORMATICA 2003
Volume 24 Number 4 (56)

Marcin GORAWSKI, Michał PIEKAREK
Politechnika Śląska, Instytut Informatyki

ROZWOJOWE ŚRODOWISKO ETL/JavaBeans

Streszczenie. Teoretycznie ekstrakcja danych jest zadaniem prostym, jednak w praktyce może okazać się, że proces ten zajmie nawet do 70% czasu przeznaczonego na stworzenie hurtowni danych. W celu zoptymalizowania tego procesu często projektuje się specjalizowaną, dostosowaną do wymagań konkretnego systemu hurtowni danych, własną aplikację ekstrakcji. W opracowaniu przedstawiono projekt i realizację środowiska rozwojowego do tworzenia aplikacji procesu ekstrakcji, które zminimalizuje nakłady czasowe związane z realizacją hurtowni danych.

Słowa kluczowe: proces ekstrakcji danych, hurtownie danych

DEVELOPMENT ENVIRONMENT ETL/JavaBeans

Summary. Theoretically data extraction process is a simple task, but practice shows that this process may take even up to 70% of the overall time destined for data warehouse (DW) realisation. Designers engaged in extraction process often use specialised software dedicated to specific warehouses. In this paper we present project and realisation of developmental environment that minimises time and efforts related with DW.

Keywords: extraction process, data warehouses

1. Wstęp

Ekstrakcja danych (zamiennie - proces ETL; ang. Extraction, Transformation and Load) jest jednym z ważniejszych etapów tworzenia hurtowni danych i stanowi znaczną część czasu projektowania i implementacji systemów decyzyjnych. Do najbardziej znanych uniwersalnych narzędzi ETL należą: DataStage (IBM), Warehouse Workbanch (Systemfabrik) oraz Warehouse Builder (Oracle) [3]. Uniwersalne narzędzia ETL nadają się znakomicie do wykonywania niezbyt skomplikowanych operacji dostępu do danych, ich filtrowania i

transformacji. W praktyce trudne procesy ekstrakcji często wymagają opracowania specjalizowanych aplikacji ETL z wykorzystaniem języków wysokiego poziomu.

2. Projekt środowiska ETL

Z praktycznych doświadczeń [1, 2] autorów wynikła decyzja o budowie rozwojowego środowiska ETL z zastosowaniem komponentów JavaBeans (ozn. środowisko ETL/JB). Komponenty środowiska ETL/JB odpowiedzialne są za pozyskiwanie, transformację oraz ładowanie danych do systemów hurtowni danych.

Poszczególne komponenty posiadają pewne wspólne cechy. Każdy z komponentów posiada pole określające nazwę, typ kolumn wejściowych i wyjściowych oraz informacje o tym, które pole wejściowe odpowiada poszczególnym kolumnom wyjściowym. Projektant aplikacji ekstrakcji określa kolejność wykonywania poszczególnych operacji nadając każdemu komponentowi kolejny, unikatowy numer (*IdComponent*). Każdy komponent udostępnia programiście także następujące metody:

- *int test()* – testuje konfigurację (np. czy dla każdej kolumny określono typ danych),
- *void prepare()* – realizuje operacje inicjujące (np. otwarcie pliku),
- *void close()* – wykonuje operacje końcowe, również wtedy, gdy wykonywanie procesu ekstrakcji spowoduje wyjątek programowy (np. zamknięcie pliku),
- *int run(...)* – wykonuje główne zadanie, wartość zwracana przez tę metodę w odpowiedni sposób steruje kolejnością wykonania poszczególnych operacji.

Wszystkie wspomniane pola i metody zostały zawarte w abstrakcyjnej klasie bazowej *ExBaseClass*. Oprócz tego należy również wyróżnić komponenty odpowiedzialne za pozyskiwanie i ładowanie danych. W tym celu zostały zaprojektowane dwa interfejsy: *ExSourceInterface* z metodą *readNext()* oraz *ExDestInterface* z metodą *writeNext()*.

W przypadku komponentów odpowiedzialnych za etap transformacji danych (filtrowanie, łączenie, sortowanie i agregacja) można wyróżnić dwa sposoby działania. Większość komponentów do poprawnego działania wymaga jedynie znajomości bieżącego rekordu danych, tak więc ciąg operacji zbudowany z takich komponentów może wykonywać zadania w „locie" (komponenty „zwykłe"). Jednak niektóre komponenty (np. sortowanie) do prawidłowego działania wymagają znajomości wszystkich danych.

W ich działaniu można wyróżnić trzy etapy:

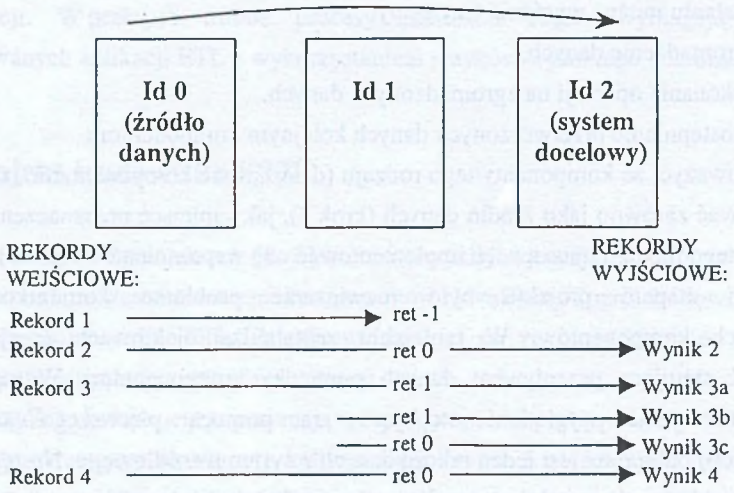
1) zgromadzenie danych,
2) dokonanie operacji na zgromadzonych danych,
3) udostępnienie przetworzonych danych kolejnym komponentom.

Łatwo zauważyć, że komponenty tego rodzaju (dalej zwane komponentami „złożonymi") można traktować zarówno jako źródło danych (krok 3), jak i miejsce przeznaczenia (krok 1). Komponenty tego rodzaju muszą więc implementować oba wspomniane wcześniej interfejsy.

Kolejnym etapem projektu było rozwiązanie problemu komunikowania się poszczególnych komponentów. W tym celu została zaprojektowana specjalna klasa *ExFramework* sterująca przepływem danych pomiędzy komponentami. W najprostszym przypadku sterowanie wygląda następująco: za pomocą pierwszego komponentu (*IdComponent* 0) pobierany jest jeden rekord danych z systemu źródłowego. Następnie rekord ten zostanie poddany odpowiednim transformacjom w kolejnych komponentach i ostatecznie załadowany do systemu docelowego (*IdComponent* n). Operacje te będą powtarzane dla każdego rekordu pobieranego ze źródła.

Jeżeli wśród komponentów transformujących dane znajdą się komponenty „złożone", to cały ciąg komponentów zostanie podzielony na kilka podciągów, a elementami łączącymi sąsiadujące podciągi będą właśnie te komponenty „złożone". Algorytm sterowania dla takiego przypadku wygląda następująco (dla uproszczenia załóżmy, że w ciągu komponentów znajduje się tylko jeden komponent „złożony" o polu *IdComponent* i): najpierw wszystkie rekordy wejściowe przetwarzane są w pierwszym podciągu (w tym wypadku jako system docelowy traktowany jest komponent *IdComponent* i), następnym etapem jest przetworzenie rekordów w drugim podciągu; tym razem rekordy pobrane z komponentu *IdComponent* i po odpowiednich transformacjach zostają załadowane do faktycznego systemu docelowego (*IdComponent* n).

Sterowanie przepływem danych pomiędzy poszczególnymi komponentami uzależnione jest od znaku wartości zwracanych za pośrednictwem metody *run()*. Gdy wartość ta jest zerowa, sterowanie przechodzi do następnego komponentu. W przypadku, gdy jest to wartość ujemna, następuje przejście do pierwszego komponentu przetwarzanego podciągu komponentów, a aktualnie analizowany rekord danych zostaje pominięty. W ten sposób realizowana jest operacja filtrowania danych. Pozostaje jeszcze trzeci przypadek, w którym zwracana jest wartość dodatnia. Oznacza to, iż po dotarciu do ostatniego komponentu aktualnie przetwarzanego podciągu klasa sterująca przekaże sterowanie do tego komponentu, który zwrócił tę dodatnią wartość. Takie zachowanie klasy sterującej jest pożądane w przypadku komponentów, które na podstawie jednego rekordu wejściowego muszą wygenerować kilka rekordów wyjściowych (np. komponent złączenia danych).



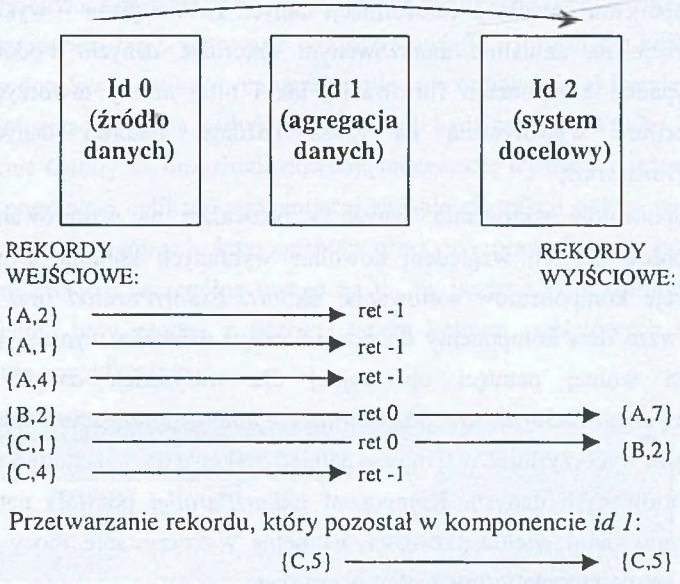
Rys. 1. Sterowanie przepływem danych
Fig. 1. Data flow control

Omówione sytuacje zobrazowane są na rys. 1. Mamy tutaj do czynienia z prostą aplikacją ekstrakcji złożoną z komponentu źródła *id 0*, komponentu przekształcającego dane *id 1* oraz komponentu docelowego *id 2*. Rekord 1 pobrany ze źródła w pierwszym obiegu pętli zostanie pominięty wskutek ujemnej wartości zwróconej przez komponent *id 1*. Rekord 2 po przekształceniach wykonanych w komponencie *id 1* zostanie zapisany do systemu docelowego. W kolejnym obiegu pętli po pobraniu ze źródła rekordu 3 komponent *id 1* zwrócił dodatnią wartość. Taka sytuacja spowodowała, że następny obieg pętli rozpoczął się od komponentu *id 1*. Dzięki takiemu zachowaniu w systemie docelowym posiadamy trzy wpisy wygenerowane na podstawie rekordu 3. Ostatni rekord ze źródła został pobrany w szóstym obiegu i po odpowiednich transformacjach został zapisany w systemie docelowym.

Przedstawione powyżej sytuacje nie wyczerpują wszystkich możliwości związanych ze sterowaniem przepływu danych. Wybór odpowiedniej ścieżki sterowania dokonywany jest na podstawie znajomości aktualnie analizowanego rekordu danych. Jednak można sobie wyobrazić sytuacje, w których potrzebna jest znajomość większej liczby rekordów.

Rysunek 2 przedstawia prostą aplikację ekstrakcji, w skład której wchodzi komponent agregacji *id 1*. Ze źródła danych zostaje pobranych sześć rekordów. W systemie docelowym mają pojawić się zsumowane wartości kolumny numerycznej. Moduł sterujący działający według omawianego do tej pory schematu pozwoli jedynie na wygenerowanie dwóch rekordów wyjściowych. Ostatnia informacja (dotycząca rekordu {C,5}) będzie dostępna dopiero wtedy, gdy zostanie zauważony brak nowych rekordów wejściowych. Aby rekord ten pojawił się na wyjściu, należy zmodyfikować algorytm działania modułu sterującego: po pobraniu ze źródła ostatniego rekordu danych i przetworzeniu go w kolejnych komponentach

moduł sterujący przegląda jeszcze raz wszystkie komponenty i wykonuje na ich rzecz metodę *doEnd()*. Komponent, który implementuje tę metodę, zostanie w tym momencie potraktowany przez moduł sterujący jako źródło danych z jednym rekordem. Spowoduje to, że ten rekord danych zostanie wysłany do pozostałych komponentów i po odpowiednich przekształceniach zapisany będzie do systemu docelowego.



Rys. 2. Sterowanie przepływem danych – przekształcanie ostatniego rekordu danych
Fig. 2. Data flow control – last data record transformation

Zaprojektowany zbiór komponentów ekstrakcji tworzy rozwojowe środowisko ETL/JavaBeans. Przez pojedyncze zadanie ekstrakcji rozumie się ciąg zadań umożliwiający pozyskanie danych z jednego źródła (pliku tekstowego lub tabeli bazy danych) i zapis tych danych do jednego miejsca przeznaczenia (pliku tekstowego lub tabeli bazy danych). W tym celu wykorzystywane są komponenty *ExSourceFile*, *ExDestFile*, *ExSourceTable* oraz *ExDestTable*. Komponenty bazodanowe pozwalają na dostęp do dowolnego serwera bazy danych przez sterownik JDBC. Komponenty *ExSourceFile* i *ExDestFile* umożliwiają odczyt i zapis informacji do plików tekstowych; dzięki tym komponentom można skorzystać z plików o stałej szerokości pól albo z plików, w których poszczególne pola rozdzielone są wskazanym znakiem alfanumerycznym.

Komponenty biorące udział w ciągu zadań ekstrakcji powinny być ponumerowane narastająco od wartości 0 (komponent pozyskania danych), kolejno, co 1 zgodnie z kolejnością wykonywanych podzadań. Komponent zapisu danych powinien posiadać maksymalny numer w ciągu. Pomiędzy tymi komponentami można umieścić dowolną liczbę komponentów transformujących dane.

Projektant może użyć:

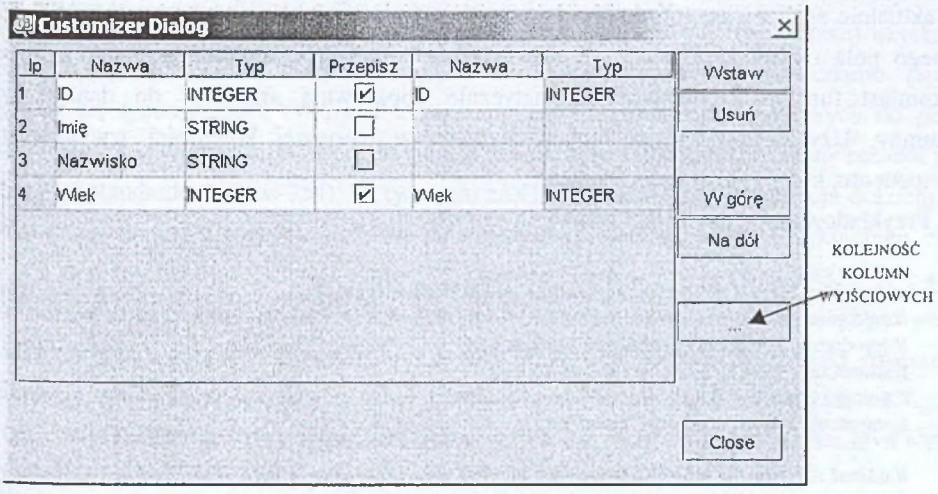
- komponentu filtrowania danych *ExFilter* – wykonuje selekcję rekordów zgodnie z warunkiem podanym w specyfikacji aplikacji. Warunek ten będzie musiał być zapisany w postaci procedury wywoływanej w trakcie wykonywania zdarzenia *OnFilter*;
- komponentu dowolnej transformacji danych *ExTransform* – wykonuje dowolne operacje na aktualnie analizowanym rekordzie danych. Podobnie jak i w przypadku komponentu filtrowania, tak i tutaj należy utworzyć odpowiednią procedurę wywoływaną na rzecz każdego rekordu danych (zdarzenie *OnTransform*);
- komponentów sortowania danych – pozwalają na posortowanie wszystkich rekordów danych względem dowolnie wybranych kolumn. Dostępne są trzy rodzaje komponentów sortowania: *ExSort*, *ExSortParallel* oraz *ExSortExtern*. Pierwsze dwa komponenty do prawidłowego działania wymagają odpowiedniej ilości wolnej pamięci operacyjnej dla wszystkich danych sortowanych. Komponent *ExSortExtern* pozwala na optymalne sortowanie bardzo dużej ilości danych wykorzystując w tym celu pamięć dyskową do przechowywania częściowo posortowanych danych. Komponent *ExSortParallel* pozwala natomiast, dzięki wykorzystaniu wielowątkowości, na pełne wykorzystanie mocy obliczeniowej, jaką dysponują platformy wieloprocesorowe;
- komponentu agregacji danych *ExAggregate* – umożliwi obliczanie agregatów dla odpowiednio pogrupowanych rekordów. Wykorzystanie mechanizmu zdarzeń pozwala na obliczenie dowolnych agregacji (dostępne są zdarzenia *OnCalculateAggregate*, *OnZeroAggregate* oraz *OnSendAggregate*). Typowe operacje (SUM, AVG, MIN, MAX i COUNT) zostały zaimplementowane w komponencie. Aby komponent ten działał prawidłowo, rekordy na wejściu muszą pojawiać się w odpowiedniej kolejności. Warunek ten spełnia umieszczenie przed komponentem *ExAggregate* jednego z komponentów sortowania, gdzie sortowania dokonujemy na tych kolumnach, względem których dokonywane będzie grupowanie;
- komponentu złączenia danych *ExJoin* – umożliwi dokonanie złączenia danych wejściowych z zewnętrznym źródłem danych. Warunkiem złączenia jest równość odpowiednio zdefiniowanych kolumn kluczowych danych zewnętrznych z odpowiednimi kolumnami rekordów wejściowych. Źródłem danych zewnętrznych może być tabela bazodanowa bądź plik tekstowy, należy jednak zagwarantować, aby wszystkie potrzebne dane zewnętrzne zmieściły się w dostępnej pamięci

operacyjnej. Dodatkowo zostało udostępnione zdarzenie *OnFilter*, dzięki któremu można dokonać wstępnej filtracji danych zewnętrznych;

- komponentu zamiany danych na podstawie zewnętrznego słownika *ExLookUp* – zasada działania podobna do operacji złączenia, gdzie w danych zewnętrznych dla konkretnego klucza zostanie wyszukany tylko jeden pasujący rekord.

Poza wymienionymi komponentami projektant ma możliwość skorzystania z komponentu *ExLogError*. Komponent ten wychwytuje wszystkie wyjątki i informacje pojawiające się w trakcie działania docelowej aplikacji oraz prezentuje je w odpowiedniej formie. Standardowo przechwycone informacje można wyświetlić na konsoli bądź zapisać do pliku. Ale można też obsłużyć zdarzenie *OnLog*, co umożliwia dowolną prezentację wybranych informacji.

Stworzenie poprawnej aplikacji wykonującej zadanie ekstrakcji polega na odpowiednim skonfigurowaniu poszczególnych komponentów oraz ich prawidłowym ponumerowaniu. Należy przy tym zwrócić szczególną uwagę na to, by liczba i typy kolumn wyjściowych danego komponentu były zgodne z liczbą i typem kolumn wejściowych komponentu o kolejnym numerze porządkowym.



Rys. 3. Edytor komponentu ExFilter
Fig. 3. ExFilter customizer

W celu łatwiejszej konfiguracji odpowiednich parametrów można skorzystać z dedykowanych edytorów właściwości, jednak to udogodnienie jest możliwe tylko wtedy, gdy korzystamy z narzędzi programistycznych wspierających technologię JavaBeans. W przeciwnym wypadku konfiguracji komponentów dokonujemy za pomocą odpowiednich funkcji ustawiających poszczególne parametry. Przykładowy edytor właściwości przedstawiony jest na rys. 3.

Środowiska programistyczne wspierające technologię JavaBeans posiadają również ułatwienia umożliwiające prostszą obsługę zdarzeń. W przypadku komponentu *ExFilter* użytkownik powinien obsłużyć zdarzenie *onFilter*. Automatycznie zostanie wygenerowany odpowiedni fragment kodu, a jedyną czynnością projektanta aplikacji jest dopisanie stosownego kodu wewnątrz funkcji *exFilter1OnFilter*:

```
try {
 exFilter1.addExFilterListener(new pl.polsl.extraction.ExFilterListener(){
     public void onFilter(pl.polsl.extraction.ExFilterEvent evt) {
       exFilter1OnFilter(evt);
     }
  });
}
catch (java.util.TooManyListenersException e) { e.printStackTrace();
}
private void exFilter1OnFilter(pl.polsl.extraction.ExFilterEvent evt) {
  // Add your handling code here:
}
```

W trakcie obsługi zdarzeń programista ma do dyspozycji funkcje umożliwiające dostęp do aktualnie analizowanego rekordu danych. W przypadku funkcji odczytujących zawartość danego pola należy zastosować jej odpowiednią wersję dla konkretnego typu kolumny. Natomiast funkcje ustawiające automatycznie dopasowują argument do danego typu kolumny. Użycie powyższych funkcji wymaga w pierwszej kolejności wyodrębnienia komponentu, który zgłosił dane zdarzenie.

Przykładowa metoda obsługi zdarzenia *OnFilter*:

```
private void exFilter1OnFilter(pl.polsl.extraction.ExFilterEvent evt) {
  // Add your handling code here:
  // wyodrębnij komponent generujący zdarzenie
  ExBaseClass ebc=(ExBaseClass)evt.getSource();
  // pobierz interesującą wartość
  Long zarobek=ebc.getInLong("placa");

  // odrzuć rekordy, dla których zarobek jest ujemny
  // i zapisz te rekordy do raportu
  if (zarobek==null || zarobek.longValue<=0) {
       evt.filter=true;
       evt.printFiltered=true;
  }
}
```

3. Wybrane aspekty budowy środowiska ETL/JavaBeans

W trakcie budowy środowisko ETL/JB było wielokrotnie testowane w celu eliminacji błędnego działania. Badano również efektywność wykonywania poszczególnych operacji.

3.1. Testowanie i uruchamianie

Testowanie prowadzono z wykorzystaniem metody mieszanej, będącej połączeniem metody wstępującej i zstępującej. Metoda wstępująca została zastosowana podczas testowania poszczególnych komponentów, natomiast metoda zstępująca – w trakcie budowy modułu sterującego. Zaletą takiego podejścia było szybkie uzyskanie w pełni działającego programu. Wraz z testowaniem nowych komponentów funkcjonalność tworzonych aplikacji wykonujących ekstrakcję zwiększała się. Kolejnym warunkiem prawidłowego testowania był dobór odpowiednich danych. Właściwy dobór danych testowych może ułatwić wykrywanie błędów, a czasem pozwala ustalić także ich źródło. Uwagę zwrócono na wartości krańcowe, a zwłaszcza sytuacje, gdy konkretne dane nie występują lub zamiast nich pojawiają się wartości *null*.

Odpowiednia kolejność tworzenia poszczególnych komponentów znacznie ułatwiła proces testowania. Jako pierwsza została stworzona klasa sterująca oraz komponenty zapisu i odczytu danych z pliku. Testowanie polegało na sprawdzeniu, czy dane są poprawnie wczytywane (bądź zapisywane), zgodnie z określonym formatem. Jeśli chodzi o efektywność dostępu do danych, to kilkunastokrotne przyspieszenie operacji odczytu (zapisu) uzyskano wprowadzając dodatkowe klasy buforujące. W trakcie testowania komponentu zapisu pojawiła się niedogodność związana z zapisem danych zmiennoprzecinkowych do pliku tekstowego: otóż w języku Java ta operacja jest czasochłonna (ale za to uzyskany rezultat jest zgodny ze standardem IEEE-754). W tym celu została napisana prosta aplikacja dokonująca wielokrotnej zamiany różnych liczb zmiennoprzecinkowych na tekst. Przykładowo czas trwania zamiany liczby 43,1 trwał około 12s, natomiast zamiana liczby 0,0000002341 trwała 63s (ponieważ czas zamiany pojedynczej liczby jest stosunkowo krótki, tak więc powyższe wyniki dotyczą zamiany wykonywanej 1,5 mln razy). Dla porównania została stworzona identyczna aplikacja w języku C i czasy trwania wynosiły ok. 1,5s, niezależnie od liczby (język C wyświetla tyle cyfr znaczących, ile zażyczy sobie programista, w języku Java wynik jest wyświetlony zawsze z najlepszą dokładnością).

Kolejnym krokiem było stworzenie komponentów dostępu do baz danych, a aplikacja testująca składająca się z komponentów odczytu i zapisu, wykonywała przepisywanie danych ze źródła (plik bądź tabela) do miejsca przeznaczenia, którym był inny plik bądź tabela. Analizie zostało poddanych wiele plików i tabel zawierających od kilku rekordów do kilkunastu tysięcy wierszy danych. Potwierdzeniem poprawności działania tych komponentów była identyczna zawartość źródła i przeznaczenia.

Następnie został stworzony komponent sortowania (*ExSort*). Została również stworzona kolejna aplikacja testująca, która za zadanie miała sprawdzenie poprawności działania tego komponentu oraz części klasy sterującej odpowiedzialnej za zarządzanie komponentami

„złożonymi". Zastosowany algorytm sortowania oparty jest na metodzie sortowania poprzez łączenie (*ang. MergeSort*) i charakteryzuje się złożonością rzędu n*log(n). Istnieją natomiast dość wyraźne różnice w czasie sortowania związane ze sposobem porównywania łańcuchów tekstowych. Dla przykładu zostały wykonane testy szybkościowe, polegające na posortowaniu siedmiu kolumn tekstowych. Plik źródłowy składał się z 29 tysięcy powtarzających się rekordów (29 różnych rekordów powtórzonych tysiąc razy). Czas sortowania takiej ilości danych wyniósł około 300ms. Jednak w sytuacji, gdy w trakcie porównywania pod uwagę była brana kolejność znaków narodowych (w tym przypadku był to alfabet języka polskiego), czas ten wzrósł do ponad 11 sekund.

W trakcie sortowania coraz to większej liczby danych ujawniły się problemy związane z nieefektywnym zarządzaniem pamięcią przez środowisko maszyny wirtualnej Java (JVM; ang. Java Virtual Machine), zwłaszcza w sytuacji, gdy Java musi operować na pamięci wirtualnej.

Aby dokładniej prześledzić sposób zarządzania pamięcią, można skorzystać z funkcji, jakie udostępnia JVM. Środowisko Java pozwala wskazać, jaką ilość pamięci może maksymalnie wykorzystać JVM oraz ile pamięci ma zaalokować na początku. Domyślnie dla JDK1.4 jest to 64MB oraz 2MB. Dodatkowo można wygenerować plik zawierający szczegółowe informacje pracy odśmiecacza pamięci (odśmiecacz – mechanizm zaimplementowany w Javie odpowiedzialny za zarządzanie i odzyskiwanie pamięci, ang. gc - *garbage collector*).

Zostały wykonane dwa testy polegające na posortowaniu zbioru danych zawierających 3,5 mln rekordów (taka ilość danych zmieści się bez problemu w dostępnej pamięci fizycznej, tj. 170MB). Podczas pierwszego testu parametry startowe JVM zostały ustawione na 380MB (maksymalny rozmiar pamięci możliwy do wykorzystania na testowanej platformie sprzętowej) oraz rozmiar początkowej alokacji 180MB; czas sortowania wyniósł w tym przypadku 64 sekundy. Drugi test różnił się wartością początkowej alokacji pamięci (tym razem 100MB), a wynik sortowania wyniósł 31 sekund. Dokładna analiza pliku raportu *gc* (*garbage collector*) pozwala na wysnucie następujących wniosków.

Java rezerwuje ciągle nowe zasoby pamięci w początkowym obszarze (180MB lub 100MB). Kiedy faktycznie zaalokowana pamięć zbliży się do tej granicy, to JVM wywołuje funkcję pełnego odśmiecania, co powoduje zwolnienie około 20MB pamięci. Ta operacja zajmuje stosunkowo dużo czasu. Ponadto JVM wykonuje „szczątkowe" odzyskiwanie pamięci, jednak czas trwania tej operacji silnie związany jest ze stanem uporządkowania pamięci; trwa ono tym dłużej, im dawniej było wykonywane pełne odśmiecanie. Dlatego też rezerwując najpierw 180MB system nie dbał o pamięć, a dopiero kiedy zbliżył się do tej granicy (co miało miejsce pod koniec działania programu), zaczął pełne odśmiecanie.

Natomiast odśmiecania „szczątkowe" wskutek nieuporządkowanego stanu pamięci trwały stosunkowo długo (czasy rzędu 0,1s). W przypadku początkowej rezerwacji 100MB pełne odśmiecanie pamięci (trwające ok. 5 sekund) nastąpiło znacznie szybciej, toteż późniejsze „szczątkowe" odśmiecania trwały krócej niż podczas wykonywania pierwszego testu.

Kolejny test polegał na posortowaniu takiej ilości danych, która nieznacznie przewyższa ilość wolnej pamięci fizycznej. W tym celu do testu użyty został plik zawierający 4,4 mln rekordów (zapotrzebowanie na pamięć w tym przypadku wynosi nieco ponad 180MB, przy dostępnej pamięci fizycznej rzędu 170MB). W tym przypadku czas sortowania wyniósł ponad 7 min, a więc widoczny był zdecydowany spadek wydajności. Podczas tego testu Java zaczęła korzystać z wirtualnej pamięci dyskowej, dlatego odśmiecanie pamięci trwało bardzo długo (z raportu *gc* wynika, iż pełne odśmiecanie trwało ponad 300 sekund); również odśmiecanie „szczątkowe", zwłaszcza obszarów znajdujących się w pamięci dyskowej, trwało także znacznie dłużej.

Ostatnim testem była próba posortowania 8 mln wierszy, przy ustawieniu parametrów JVM na ilość 380MB dostępnej pamięci (program do zakończenia potrzebował 340MB pamięci). Rezultat został otrzymany dopiero po trzech godzinach, a jeden przebieg pełnego odśmiecania pamięci trwał (na podstawie raportu *gc*) prawie dwie godziny. Ostatecznie wnioski dotyczące Javy są następujące: *JAVA działa bardzo wolno, jeżeli musi korzystać z pamięci wirtualnej; najlepsza sytuacja jest wtedy, gdy wszystkie potrzebne dane mieszczą się w pamięci fizycznej, a początkowy rozmiar pamięci ustawiony jest na niezbyt dużą wielkość.*

Z tego też względu wynikła konieczność stworzenia komponentu sortowania wykorzystującego zewnętrzne pliki tymczasowe; pozwoliło to na znaczne skrócenie czasu sortowania danych nie mieszczących się w pamięci operacyjnej. Dla przykładu czas posortowania 40 mln rekordów z wykorzystaniem komponentu *ExSortExtern* wynosi tylko 35 minut.

3.2. Komponenty sortowania

Przeprowadzone testy wskazały, że jednym z najbardziej czasochłonnych zadań jest sortowanie dużej liczy danych. Spowodowało to poszukiwanie nowych, bardziej wydajnych metod sortowania, które skracają czas potrzebny na wykonanie tej operacji. Efektem tych poszukiwań są opisane poniżej komponenty *ExSortParallel* oraz *ExSortExtern*.

3.2.1. Sortowanie równoległe

Aby przyspieszyć proces sortowania coraz częściej wykorzystuje się platformy wieloprocesorowe, jednak aby można było w pełni wykorzystać możliwości takiego sprzętu, należy odpowiednio zaprogramować algorytm sortowania. Standardowy komponent *ExSort*

jest przystosowany do środowiska jednoprocesorowego, tak więc wynikła konieczność stworzenia nowego komponentu, który będzie mógł w pełni wykorzystać dostępne możliwości sprzętowe.

Komponent *ExSortParallel* z jednej strony powinien wykorzystywać wielowątkowość (gdyż tylko programy wielowątkowe mogą jednocześnie wykorzystywać wiele procesorów), z drugiej natomiast powinien być tak zaprojektowany, aby spełniał wymogi modułu sterującego.

Pierwszym krokiem jest przygotowanie danych dla poszczególnych wątków. Każdy z wątków otrzymuje prawie identyczną ilość rekordów, a ponieważ liczba danych przeznaczonych do sortowania nie jest znana z góry, tak więc informacje pojawiające się na wejściu są cyklicznie rozdzielane do buforów poszczególnych wątków.

Dopiero wtedy, gdy wszystkie rekordy wejściowe zostaną odpowiednio rozdzielone do wątków, następuje proces sortowania poszczególnych buforów danych. Ostatnim krokiem, który należy wykonać, jest scalenie uzyskanych informacji, aby uzyskać faktycznie posortowany zbiór wszystkich rekordów wejściowych. Specyfika działania modułu sterującego znacznie ułatwia zrealizowanie tego kroku. Nie jest nam potrzebny naraz cały zbiór posortowanych danych, wystarczy jedynie znajomość kolejnego rekordu, który należy wysłać dalej.

Znając sposób realizacji sortowania równoległego można obliczyć średnie przyspieszenie realizacji tego zadania w stosunku do czasu działania komponentu *ExSort*.

Algorytm sortowania dostarczony w standardowym pakiecie Java posiada średnią złożoność rzędu $n \log n$. Zakładając, iż mamy dostęp do platformy k-procesorowej, to przy wykorzystaniu komponentu *ExSortParallel* możemy określić:

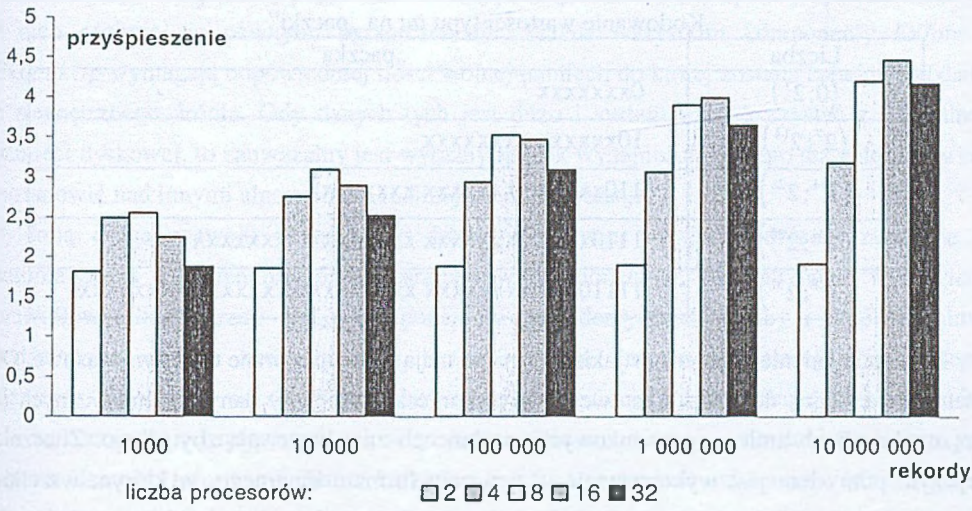
1) czas samego sortowania paczek danych: $\frac{n}{k} \log \frac{n}{k}$,
2) czas złączenia posortowanych paczek:
   - początkowe sortowanie (wysłanie pierwszej danej): $k \log k$,
   - wysłanie pozostałych n-1 informacji: $(n-1)\log k$, gdzie czynnik logarytmiczny określa złożoność wyszukiwania binarnego.

Ostatecznie więc złożoność komponentu ExSortParallel wynosi (1):

$$\frac{n}{k} \log \frac{n}{k} + k \log k + (n-1)\log k = \frac{n}{k} \log n + (n + k - 1 - \frac{n}{k}) \log k \qquad (1)$$

Widać wyraźnie, że zastosowanie platformy k-procesorowej nie przekłada się na k-krotne przyspieszenie operacji sortowania. To co zostało zyskane dzięki zrównolegleniu operacji sortowania poszczególnych paczek, jest w części tracone w trakcie złączania uzyskanych

danych. Co więcej, nie zawsze zwiększenie liczby procesorów pozwoli na przyspieszenie sortowania.



Rys. 4. Przyśpieszenie operacji sortowania
Fig. 4. Acceleration multi-thread sorting

Można również pokazać, co się dzieje w momencie, gdy uruchamiamy proces wielowątkowego sortowania na platformie jednoprocesorowej. W tym wypadku zwiększa się k-krotnie czas sortowania wszystkich paczek danych (2):

$$n\log\frac{n}{k} + k\log k + (n-1)\log k = n\log n + (k-1)\log k \qquad (2)$$

Uzyskany wynik jest gorszy niż w przypadku zastosowania standardowego komponentu sortowania *ExSort*. W rzeczywistości różnica działania jest jeszcze większa, gdyż w powyższych obliczeniach nie zostały uwzględnione narzuty czasowe związane z przełączaniem pomiędzy poszczególnymi wątkami.

3.2.2. Sortowanie zewnętrzne – format pliku tymczasowego

Mając na uwadze przeprowadzone testy, kolejnym problemem związanym z sortowaniem jest zbyt mała ilość pamięci operacyjnej, w której mogłyby się zmieścić wszystkie potrzebne dane, a wykorzystywanie w tym celu pamięci wirtualnej zarządzanej przez system operacyjny powoduje znaczny spadek efektywności przeprowadzanych operacji. Warto więc zastanowić się nad innym sposobem przechowywania dużej ilości informacji, przykładowo może to być przechowywanie posortowanych danych w zewnętrznych plikach. Oczywiście i algorytm sortowania musi być adekwatny do tej sytuacji; podobnie jak i dla sortowania równoległego,

tak i tutaj można wyróżnić etapy sortowania poszczególnych fragmentów danych oraz etap ich łączenia, z tą różnicą, że poszczególne posortowane fragmenty będą zapisane na dysku.

Tabela 1

Kodowanie wartości typu *int* na „paczki"

Liczba	„paczka"
$\langle 0;2^7)$	0xxxxxxx
$\langle 2^7;2^{14})$	10xxxxxx xxxxxxxx
$\langle 2^{14};2^{21})$	110xxxxx xxxxxxxx xxxxxxxx
$\langle 2^{21};2^{28})$	1110xxxx xxxxxxxx xxxxxxxx xxxxxxxx
$\langle 2^{28};2^{31})$	11110000 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

Pozostaje jedynie problem, w jakim formacie mają być zapisywane dane tymczasowe. Od razu odpada zapis do pliku tekstowego, gdyż konieczne zamiany danych numerycznych (a szczególnie liczb zmiennoprzecinkowych) na łańcuch znaków trwają zbyt długo. Znacznie lepszym pomysłem jest wykorzystanie w tym celu formatu binarnego, w którym wszelkie dane numeryczne zapisywane są w postaci bitowej. Pozostaje jeszcze problem związany z oznaczaniem wartości pustych *null*. Nie można z góry założyć, iż *null* odpowiada wartości zerowej bądź łańcuchowi pustemu. Przyjęty został więc następujący format zapisu jednego rekordu danych: pierwsza „paczka" określa, ile kolumn dla tego rekordu jest wartością pustą, jeśli są jakieś puste dane, to następne „paczki" zawierają numery tychże kolumn (kolumny numerowane są od zera); jako ostatnie są zapisane wartości kolejnych niepustych kolumn. Określenie „paczka" oznacza tutaj ciąg bajtów danych, za pomocą których zakodowana jest liczba typu *int* (teoretycznie kolumn może być 2^{31}-1), a sposób kodowania przedstawiony jest w tab. 1. Wartości poszczególnych niepustych kolumn zapisywane są w postaci zgodnej z typem danej kolumny. I tak dane całkowite (*long*) oraz rzeczywiste (*double*) zapisywane są z wykorzystaniem ośmiu bajtów. Nieco inna sytuacja dotyczy typu łańcuchowego. W języku Java kody znaków są 16-bitowe, jednak w większości przypadków wykorzystywany jest standardowy zestaw pierwszych 127 znaków. Dlatego też pakiet Java udostępnia możliwość zapisu (i odczytu) łańcuchów *String* w formacie *UTF* (poszczególne znaki mogą zajmować jeden, dwa bądź trzy bajty, natomiast pierwsze dwa bajty określają, ile bajtów zajmuje dany łańcuch po sformatowaniu go zgodnie z algorytmem *UTF*). Dokładny opis tego formatu można znaleźć w dokumentacji Javy[1] [4]. Na potrzeby tej pracy został zastosowany zmodyfikowany algorytm *UTF*, w którym długość sformatowanego łańcucha zapisywana jest nie na dwu bajtach, lecz w postaci wcześniej opisanej „paczki".

[1] Dokumentacja dostępna jest pod adresem: **http://java.sun.com/j2se/1.4/index.html**

4. Podsumowanie

Stworzone komponenty poprawnie realizują zaimplementowane operacje, jednak niektóre z nich cechują się pewnymi ograniczeniami. Przede wszystkim komponenty *ExJoin* i *ExLookUp* wymagają odpowiedniej ilości wolnej pamięci, do której zostaną załadowane dane z zewnętrznego źródła. Gdy danych tych jest dużo i system musi korzystać z wirtualnej pamięci dyskowej, to zauważalny jest wyraźny spadek wydajności. Dlatego też należałoby się zastanowić nad innymi algorytmami realizującymi te operacje.

Inną drogą rozwoju stworzonego środowiska mogą być udogodnienia związane z automatyzacją ustawień niektórych parametrów komponentów. Podstawowym wymogiem prawidłowej konfiguracji ciągu komponentów jest identyczność liczby i typów kolumn wyjściowych danego komponentu z liczbą i typem kolumn wejściowych następnego elementu. W obecnej wersji edytory ułatwiające konfigurację poszczególnych komponentów nie wykorzystują tej informacji, tak więc programista zmuszany jest do dublowania raz już wprowadzonych informacji. Zatem korzystne byłoby rozwiązanie uniwersalne (działające w dowolnym środowisku rozwojowym Java), dzięki któremu możliwe byłoby automatyczne ustawianie kolumn na podstawie parametrów poprzedniego (lub następnego) komponentu tworzącego dany ciąg zadań.

LITERATURA

1. Gorawski M.: Systemy Wspomagania Podejmowania Decyzji. Software 5/1999
2. Gorawski M., Koziatek A.: Ekstrakcja danych. Software 9/1999
3. Gorawski M., Świtoński A.: Uniwersalne narzędzia ETL. Informatyka 10/2000
4. Java™ 2 SDK, Standard Edition, Documentation, Sun Microsystems, 2002.
5. JavaBeans™ API Specification, Sun Microsystems, 1997.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 30 czerwca 2003 r.

Abstract

This paper presents ETL/JavaBeans (ETL/JB) environment that makes easier process of building new ETL application. Based on our practical experience, we have decided to create components that are been responsible for extraction, transformation and data loading operations. There are *ExSourceFile*, *ExSourceTable* *ExAggregate*, *ExFilter*, *ExJoin*, *ExLogError*, *ExLookUp*, *ExSort*, *ExSortExtern*, *ExSortParallel*, *ExTransform*, *ExDestFile*, and *ExDestTable* components.

Next we have concentrated on the communication between components. In this effect we have created a special class *ExFramework* that controls data flow. During the development of ETL/JB environment realisation we performed many tests. We also have tested the efficiency of ETL process.

The components that were presented in this paper properly perform ETL process, but some of them have restrictions. First of all, *ExJoin* and *ExLookUp* components require a lot of system memory to store all data from external source. If these data are huge, computer will use virtual memory. As a result we will notice decrease in performance. That is why we are going to apply better algorithms.

Additionally, we are going to simplify components configurations, especially we may use the fact that the number and types of output columns must be identical with the number and types of input columns of subsequent component. In this ETL/JB environment designer must duplicate the same information many times. This behaviour is forced by JavaBeans specification so we are going to develop software that will be dedicated to ETL/JB components.

Adresy

Marcin GORAWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, M.Gorawski@zti.iinf.polsl.gliwice.pl.

Michał PIEKAREK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, M.Piekarek@zti.iinf.polsl.gliwice.pl.