STUDIA INFORMATICA 2006
Volume 27 Number 1 (66)

Marcin GORAWSKI, Mariusz CIEPLUCH
Politechnika Śląska, Instytut Informatyki

PRZYROSTOWA EKSTRAKCJA DANYCH ETL(δ)

Streszczenie. W celu poprawienia dostępności danych do analiz w hurtowniach danych zaproponowano rozbudowę funkcjonalności eksploatowanego systemu ETL o proces przyrostowej ekstrakcji danych źródłowych (δ). Taki system ETL(δ) pozwala zachować pełną historię zmian w danych początkowych, a aktualizacja może odbywać się równolegle z dostępem do danych.

Słowa kluczowe: hurtownie danych, ETL, ekstrakcja, ciągła integracja danych

INCREMENTAL DATA EXTRACTION ETL(δ)

Summary. To aquire a goal of increasing data availability for data analysis in data warehouses authors propose functionality build-up of exploited ETL systems by adding incremental source data (δ) extraction process. Such ETL(δ) system allows keeping of full history of changes made in source data, and actualization can be made in parallel with data access.

Keywords: data warehouse, ETL, extraction, continuous data integration

1. Wstęp

Obecnie dostęp do aktualnych danych w jak najkrótszym czasie staje się kluczowym zagadnieniem projektowanych systemów hurtowni danych. Przyrostowa ekstrakcja danych jest jedną z cech hurtowni danych czasu rzeczywistego, dla których określony jest maksymalny, stosunkowo krótki czas propagacji zmian w źródłach danych. Oczywiście, trudno mówić tutaj o czasach rzędu milisekund, ale rząd minut jest rozsądną granicą do osiągnięcia. Wiadomo, sama wartość graniczna może być różna dla różnych zastosowań.

W pracy [1] podjęto próbę analizy tego problemu. Autorzy zaproponowali trójwarstwową architekturę procesu ETL, gdzie każda warstwa jest funkcjonalnym rozwinięciem odpowiadającej jej części procesu. Dane w systemie ETL były reprezentowane w języku XML, połączenia pomiędzy warstwami odbywały się bezpośrednio, poprzez szybkie złączki, bez składowania danych pośrednich w plikach. System ten zaimplementowano w środowisku J2EE ETL w formie zbioru kontenerów, usług i adapterów, bez udokumentowanych praktycznych wyników.

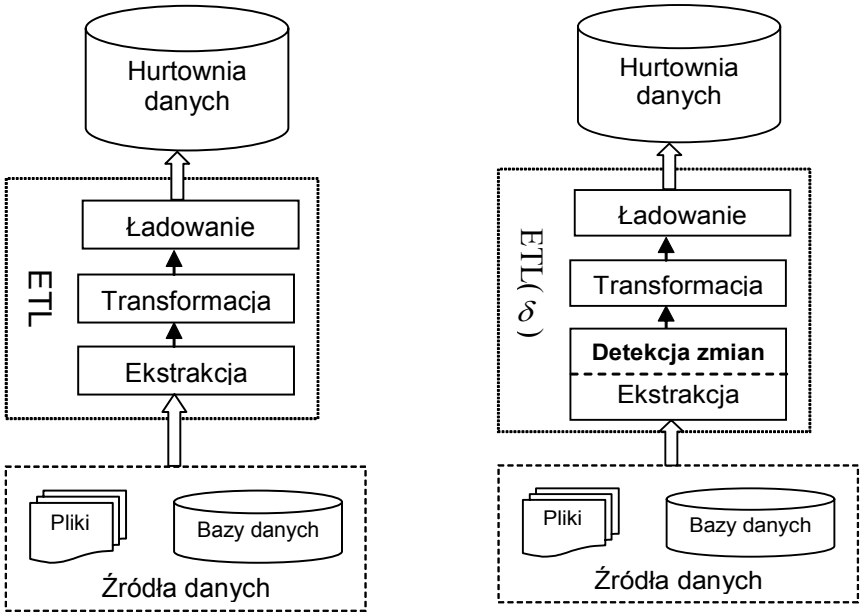
W pracy [2] przedstawiono pewne definicje i problemy realizacji hurtowni danych czasu rzeczywistego. W głównej mierze zarysowano dwa ważne aspekty problemu: współbieżność aktualizacji (aktualizacja danych wraz z równoległym dostępem do danych) oraz czas aktualizacji samych danych w hurtowni, wydłużony o aktualizację wszelkich struktur wzmacniających wykonanie analiz danych.

2. Proces ETL(δ)

Prezentowany system ETL(δ) uwzględnia w klasycznym procesie ekstrakcji, transformacji i ładowania danych (ang. *Extraction, Transformation, Load – ETL*) proces przyrostowej aktualizacji danych δ dzięki etekcji zmian w danych źródłowych. Takie podejście jest kontynuacją wcześniejszych badań własnych [3-13] i kolejnym krokiem w realizacji ciągłej integracji danych.

W klasycznym procesie ETL ekstrakcja danych następuje zwykle do pustej struktury hurtowni danych. W takim przypadku, jeśli chcemy zaktualizować zawartość hurtowni, należy cały proces ETL wykonać od nowa. Takie rozwiązanie obarczone jest stratą czasu na ponowną ekstrakcję oraz utratę historii zmian w danych. Podczas każdej reekstrakcji dodatkowo dane w hurtowni są niedostępne dla analiz. Po uwzględnieniu masowości danych w hurtowniach ciągły proces reekstrakcji i utrata historii zmian obniżają znacznie wartość danych oraz zwiększają koszt ich uzyskania.

W procesie ETL(δ) dodajemy proces detekcji zmienionych danych źródłowych (rys. 1) na samym początku całego procesu i operujemy tylko na samych danych zmienionych [14]. Pozwala to na przyrostowe aktualizowanie danych w hurtowni, z zachowaniem historii zmian, co przy dobrej współbieżności zapewnia ciągłą dostępność danych. Właśnie ta ostatnia cecha, czyli ciągła dostępność danych, jest pożądana dla procesu ETL(δ).



Rys. 1. Różnice pomiędzy procesami ETL (po lewej) i ETL(δ) (po prawej)
Fig. 1. Differences between ETL (on the left) and ETL(δ) (on the right) processes

2.1. Detekcja zmian w danych początkowych

Proces detekcji zmian w danych początkowych jest złożony, a sprawne wykonanie detekcji jest podstawą szybkiego procesu ETL(δ).

Optymalny jest bezpośredni dostęp do samych zmienionych danych, gdyż ciężar detekcji zmian jest przerzucany na źródło danych. Jedną z możliwości jest przerzucenie ciężaru detekcji zmian na stronę systemu zarządzania danymi (RDBMS). Jeśli tylko baza danych udostępnia taką możliwość, można wykorzystać ją do detekcji zmian w danych, poprzez zdefiniowanie odpowiednich wyzwalaczy na każdą z operacji modyfikacji: wstawienie/aktualizację/usunięcie. Jeśli w tak stworzonych wyzwalaczach zawrze się np. kopiowanie z odpowiednią flagą danych zmienianych do odpowiedniej tablicy tymczasowej, to zostaną w niej zawarte same krotki zmienione. Tak więc proces ETL(δ) jest usprawniony, bowiem wystarczy pobrać i wykasować dane tylko z tej tablicy.

Jeśli w bazie jest przechowywany czas operacji zmiany każdej krotki (np. w bazach czasowych), możemy zauważyć, że przez wykonanie zapytania czasowego możemy uzyskać wszystkie dane zmienione.

Większość RDBMS można skonfigurować, aby wykonywane operacje były zapisywane do pliku dziennika. Tak więc na podstawie analizy takiego pliku można odtworzyć historię zmian. Mając stan początkowy, odtwarzając listę zmian zapisaną w pliku dziennika, można znaleźć dane, które się zmieniły.

Nie wszystkie bazy danych mają możliwość aktywnego raportowania o zmianach, czy to poprzez wykorzystanie wyzwalaczy, czy zapis do pliku dziennika w sposób możliwy do jednoznacznej interpretacji.

Detekcja zmian oparta na porównaniu migawek wymaga sprawnej metody porównania pojedynczych krotek danych. Aby wykryć krotkę zmienioną, można porównywać poszczególne kolumny sekwencyjnie, i jeśli chociaż jedna jest różna, to wartość takiej krotki została zmieniona. Takie porównanie ma jednak swoje wady, w szczególności gdy rozpatrujemy relacje o dużej liczbie kolumn, czy z dużą liczbą danych znakowych. W takim przypadku znacznie wzrasta złożoność detekcji zmian. Alternatywną propozycją jest wykorzystanie do detekcji zmian znaczników, obliczanych na każdej pojedynczej krotce danych [1, 2]. Sprowadza to proces detekcji zmian do prostego porównania pojedynczych wartości całkowitych. Na koniec wszystkie dane, które nie zostały porównane (odpytane) po stronie migawki poprzedniej, są danymi usuniętymi, a po stronie migawki aktualnej są danymi dodanymi. Tak więc przez sekwencyjne przejrzenie dwóch zbiorów można wykryć zmiany w danych początkowych.

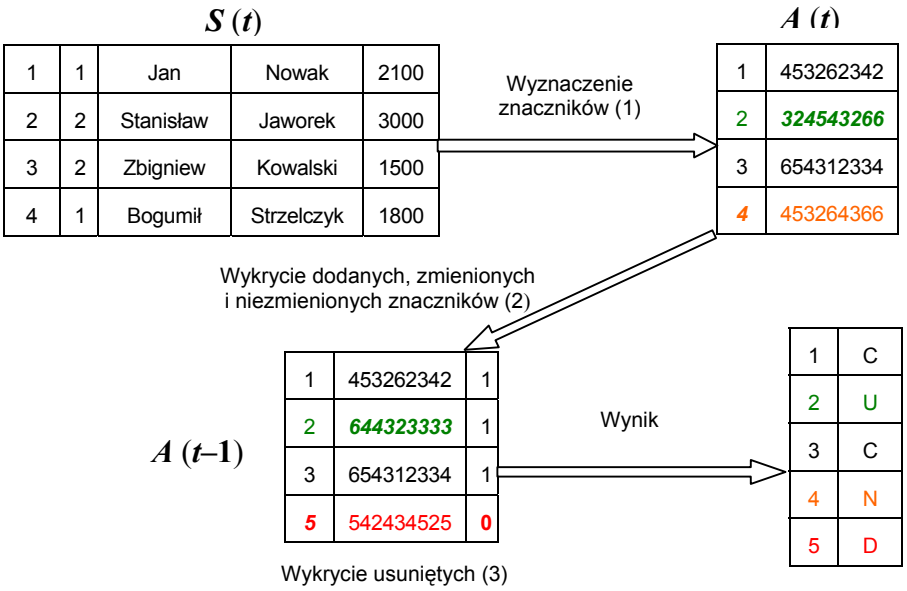
Oczywiście, do poprawnej identyfikacji krotki trzeba zachować osobno klucz każdej krotki wraz z jej znacznikiem. Przyjmując, że zbiór $\boldsymbol{A}(\boldsymbol{t}\text{-}1)$ jest zbiorem par: *klucz krotki – skrót* reprezentujący migawkę *PRZED*, a zbiór $\boldsymbol{A}(\boldsymbol{t})$ reprezentujący migawkę *PO* oraz zbiór $\boldsymbol{S}(\boldsymbol{t})$ reprezentujący tablicę źródłową, to algorytm porównania dwóch migawek (przed i po) oparty na znacznikach można przedstawić następująco:

1. *Wyznacz nowy zbiór znaczników* ***A(t)***
    a. *Dla każdej krotki t ze zbioru* ***S(t)*** *wyznacz jej skrót*
        i. *Do zbioru* ***A(t)*** *dodaj parę (id_krotki(t), skrót(t))*
2. *Dla każdej krotki t (id, skrót) z* ***A(t)***
    a. *Zapytaj zbiór* ***A(t–1)*** *o klucz id*
        i. *Jeśli jest odpowiedź, to porównaj uzyskany wynik ze skrótem*
            1. *jeśli jest równy, to krotka jest niezmieniona*
            2. *jeśli jest różny, to krotka jest zmieniona*
            3. *odznacz, że dana krotka została odpytana*
        ii. *Jeśli nie ma odpowiedzi, to krotka jest nowa*
3. *Wykryj usunięte*
    a. *Przejrzyj zbiór* ***A(t–1)*** *i wykryj wszystkie krotki, które nie zostały odpytane w kroku 2.a*
4. *Zapisz* ***A(t)*** *jako nową* ***A(t–1)***

Wykorzystywanie znaczników do detekcji zmian w danych wymaga ustalenia akceptowalnego stosunku niejednoznaczności (takie same wartości skrótu dla dwóch różnych

danych) do czasu obliczania skrótu krotki. Jeśli chcemy pójść w stronę szybko obliczanych skrótów, należy się liczyć z większym prawdopodobieństwem powtórzenia skrótu. Jednak ten procent powinien być nadal mały, a utrata poniżej 1% danych w hurtowni danych może być do zaakceptowania. Można wyróżnić dwa możliwe zastosowania prostych wyznaczeń skrótów każdej krotki, które zmniejszają prawdopodobieństwo powtórzenia wartości, a mianowicie:

- Pozycyjne wyznaczenie funkcji skrótu, gdzie wartość skrótu krotki jest sumą wartości skrótu każdej kolumny mnożoną przez kolejną liczbę pierwszą, tak aby każda kolumna miała inny współczynnik.
- Dwuetapowy proces obliczania skrótu. W takim przypadku mamy N prostych funkcji obliczania skrótu. Obliczenie skrótu jest dwuetapowe: (1) wylosowanie wykorzystywanej funkcji skrótu oraz (2) na podstawie wybranej funkcji obliczenie wartości skrótu. Zatem, prawdopodobieństwo powtórzenia wartości maleje w przybliżeniu N razy (dokładnie (N–γ) razy, gdzie γ jest współczynnikiem powtarzalności dwóch wartości obliczonych za pomocą dwóch różnych funkcji).



Rys. 2. Przykład detekcji zmian opartej na znacznikach
Fig. 2. Example of change of data based on signatures

Na rys. 2 pokazano przykład wyznaczenia zmian za pomocą porównywania migawek opartego na znacznikach. Zgodnie z przedstawionym powyżej algorytmem mamy dwie tablice (migawki) znaczników: $A(t\text{-}1)$ (PRZED) i $A(t)$ (PO) oraz nową tablicę źródłową $S(t)$ do sprawdzenia zmian. W kroku (1) wyznaczamy nowe wartości skrótów przybyłych krotek, w kroku (2) porównujemy klucze i znaczniki z tablicy $A(t)$ z wartościami w tablicy $A(t–1)$ w celu wykrycia krotek zmienionych i dodanych. W kroku (3) wykonujemy odszukanie

wartości o nieustawionej fladze odpytania w celu wykrycia usuniętych krotek. Na wyjściu mamy tablicę wykrytych zmian.

3. System ETL(δ)

System ETL(δ) zaimplementowany w języku Java jest modyfikacją już istniejącego systemu ETL [11]. Dla prawidłowego jego funkcjonowania wprowadzono kilka zmian:

- Definiowania typu krotki, aby odzwierciedlić wynik detekcji. Każda krotka może być jednego z czterech typów: nowa (N), zmieniona (U), usunięta (D), bez zmian (C). Każdy typ odzwierciedla rodzaj modyfikacji danej krotki od czasu poprzedniej detekcji.
- Przeprojektowania komponentów tak, aby operowały na typach krotek.
- Dodania komponentu detekcji zmian, który wykrywa zmiany w danych źródłowych i nadaje typ każdej krotce.
- System ETL(δ) implementuje przyrostową aktualizację docelowych tablic. Na podstawie wykrytych danych zmienionych przeprowadza, w oparciu o nie, częściową transformację i tworzy docelowe zmiany w tabelach. Te zmiany po stronie insertorów są o określonym rodzaju (N, D lub U). Tak więc wdrażamy je odpowiednio wg dwóch możliwych scenariuszy: 1) krotki typu U i D są wstawiane jako nowe, ale nadpisują odpowiednie odniesienie do poprzednich ich wartości; 2) krotki typu U nadpisują poprzednie wartości, a krotki D są usuwane z systemu. Krotki typu N są zawsze wstawiane do systemu. Jak widać, możemy sterować przechowywaną historią i zarządzać nią po stronie insertorów w zależności od wybranego scenariusza aktualizacji tabel docelowych. W obecnej wersji systemu przyjęto dla uproszczenia drugi sposób.

Niektóre komponenty transformacji dla prawidłowego działania muszą znać pozostałe dane, dlatego mają lokalne repozytoria przetwarzanych danych. Dla przykładu, komponent grupowania musi znać rozkład wszystkich dotychczasowych grup, aby móc poprawnie dopasować przybyłą krotkę do już istniejącej grupy, albo stworzyć dla niej nową grupę. Komponenty wymagały pewnych modyfikacji związanych z występowaniem znaczników, bowiem przetwarzanie opiera się tylko na danych zmienionych (krotkach typu N, U, D).

Komponent detekcji zmian

W celu zachowania uniwersalności komponent detekcji zmian został opracowany jako osobny element. Przy takim rozwiązaniu jest niezależny od tego, jaki ekstraktor będzie go poprzedzał. Jego zadaniem jest wykrycie danych zmienionych na podstawie otrzymanych danych i swojego repozytorium danych, a następnie wysłanie ich z odpowiednimi znacznikami N, U i D.

Komponent detekcji zmian porównuje migawki oparte na znacznikach. W wyniku tego porównania zostaje wyznaczony typ każdej krotki, a krotki zmienione (N, U, D) są wysyłane dalej. Dodatkowo, dla poprawnego działania systemu krotki typu U i D są uzupełniane poprzednimi wartościami z lokalnego repozytorium wartości poprzednich, które są osobno przechowywane przez komponent w plikach danych. Osobne przechowanie wartości krotek i skrótów służy do przyspieszenia operacji poprzez uniknięcie wyliczania skrótów dla poprzednich wartości. Przykład przechowywanych danych został przedstawiony na rys. 3.

A) Poprzednie wartości krotek

1	1	Jan	Nowak	2100
2	2	Stanisław	Jaworek	3000
3	2	Zbigniew	Kowalski	1500
4	1	Bogumił	Strzelczyk	1800

B) Skróty krotek

1	453262342
2	324543266
3	654312334
4	453264366

Rys. 3. Przykład danych przechowywanych przez komponent detekcji zmian
Fig. 3. Example of data store by component of change detection

Komponenty ekstrakcji danych

Wyróżniono dwa komponenty: ekstrakcji z pliku tekstowego i z bazy danych. Oba są ekstraktorami biernymi i dla prawidłowego ich funkcjonowania zaraz za nimi powinien się znaleźć komponent detekcji zmian. Funkcjonalność ekstraktorów sprowadza się do sekwencyjnego pobierania danych ze źródła (kolejnych linii z pliku, czy wierszy z tabeli w bazie danych), opakowania ich w obiekt krotki i wysłanie do bufora wyjściowego.

Komponenty transformacji danych

Komponenty transformacji danych podzielono na komponenty zależne i niezależne od danych wcześniejszych. Komponenty zależne wymagają przechowywania wszystkich swoich pośrednich wyników oraz ich aktualizacji wraz z napływem kolejnych krotek. Komponenty niezależne nie wymagają znajomości danych, tylko przetwarzają każdą krotkę w locie.

Do komponentów zależnych od danych możemy zaliczyć:

Komponent *Agregacji* wykonuje zestaw funkcji agregacji (jedna z pięciu funkcji: SUM, AVG, COUNT, MIN, MAX) na wszystkich danych przechodzących przez komponent. Dane funkcje agregacji są rozdzielcze, więc nie ma problemu z aktualizacją wyniku. Dla prawidłowego funkcjonowania komponent musi przechowywać aktualne wartości poszczególnych agregatów pomiędzy kolejnymi etapami procesu. W zależności od wykonywanej funkcji agregacji i typu otrzymanej krotki są wykonywane różne operacje. Dwie pierwsze funkcje (SUM i COUNT) są proste i jednoznaczne: w zależności od typu poprawiamy sumę lub liczebność krotek wg ustalonego wzoru. W przypadku funkcji średniej (AVG) musimy

znać sumę i liczebność, aby móc poprawnie określać średnią wraz z przepływem kolejnych danych.

Tabela 1

Komponent agregacji – przechowywane dane i wykonywane operacje

Funkcja	**Przechowywane wartości**	**Typ krotki**		
		N	**U**	**D**
SUM	Suma (*sum*)	*sum = sum + val*	*sum = sum + NEWval* *sum = sum – OLDval*	*sum = sum – val*
COUNT	Liczebność (*count*)	*count = count + 1*		*count = count – 1*
AVG	Suma i liczebność (*sum*, *count*)	*count = count + 1* *sum = sum + val*	*sum = sum + NEWval* *sum = sum – OLDval*	*count = count – 1* *sum = sum – val*
MIN	Lista N ostatnich wartości minimalnych	Dodaj do listy, jeśli mniejsza od ostatniego na liście lub lista nie jest pełna	Usuń z listy, jeśli tam jest, poprzednią wartość i dodaj nową, jeśli się kwalifikuje. Za wąska lista nie kwalifikuje dodania.	Usuń z listy, jeśli się w niej zawiera (mniejsza od ostatniego na liście)
MAX	Lista N ostatnich wartości maksymalnych	Dodaj do listy, jeśli większa od ostatniego na liście lub lista nie jest pełna		Usuń z listy, jeśli się w niej zawiera (większa od ostatniego na liście)

W przypadku funkcji ekstremalnych, tj. minimum i maksimum (MIN i MAX), pojawia się problem z odtworzeniem wartości po wykasowaniu aktualnej wartości ekstremalnej. Dlatego funkcje te przechowują N ostatnich wartości ekstremalnych, dzięki czemu poza przypadkiem pesymistycznym, w którym w jednym etapie przychodzi N operacji wykasowania aktualnie przechowywanych wartości ekstremalnych, jesteśmy w stanie określić nowe ekstremum. W tabeli 1 zebrano operacje wykonywane w komponencie agregacji. Przyjęto, że *val* jest wartością agregatu aktualnie agregowanej krotki, a *NEWval* i *OLDval* są odpowiednio nową i poprzednią wartością agregatu dla zmodyfikowanych krotek.

Komponent *Grupowania* wykonuje zestaw funkcji agregacji, ale osobno w każdej znalezionej grupie. W przypadku tego komponentu musimy znać wartość każdego agregatu w każdej grupie z osobna. Dlatego dla każdej wyznaczonej grupy przechowujemy osobną tablicę wyników wykonywanych agregatów, które są aktualizowane wraz z nadchodzeniem kolejnych krotek. Drugą różnicą w grupowaniu jest to, że na wyjściu pojawiają się inne dane niż na wejściu. Wynikowe krotki są złożeniem klucza grupy i aktualnych dla niego wartości agregatów, a znacznik wynikowej krotki jest uzależniony od tego, czy dana grupa została zmodyfikowana (np. doszła nowa krotka(i) do danej grupy, wtedy ma znacznik U), czy dodana (dopiero w tym etapie przyszły krotki do danej grupy, wtedy ma znacznik N), lub usunięta (krotki, które nadeszły wyzerowały tablice agregatów, tak więc grupa została wykasowana i ma znacznik D).

Komponent *Złączenia* dopasowuje krotki z dwóch wejść na podstawie klucza złączenia. W tym wypadku też musimy przechowywać aktualny stan krotek na każdym z wejść, aby móc w każdej chwili dopasować krotkę z wejścia drugiego do danej krotki otrzymanej

z wejścia pierwszego. Kolejność złączenia jest określana przez konfigurację komponentów. Dodatkowo, na każdym etapie przechowujemy listę krotek przybyłych w danym etapie, które nie znalazły dopasowania, jeśli są z wejścia pierwszego oraz wszystkie przybyłe w danym etapie krotki z wejścia drugiego. Każdorazowe przybycie krotki na wejście powoduje wykonanie następujących czynności: (1) aktualizację wpisu w pliku danych, (2) dodanie do listy krotek przybyłych w danym etapie, gdzie przechowuje się klucz i skrót krotki oraz jej typ, a na koniec (3) próba złączenia krotki, czyli odszukanie pasującej krotki(ek) (o tym samym kluczu złączenia) z sąsiedniego wejścia. W trakcie działania komponentu *Złączenie* wyróżnia się dwa typy złączenia: (1) *warunkowy*, gdy dane na drugim wejściu ciągle napływają. W takim przypadku nie można łączyć danej z wejścia pierwszego z drugim wejściem, dopóki nie ma pewności, że dana krotka jest aktualna. Dlatego koduje się w liście fakt, że krotka przybyła w danym momencie, a korzysta się z tego podczas złączenia warunkowego, gdy ten znacznik jest ustawiony; (2) *bezwarunkowy*, gdy dane na drugim wejściu już nie napływają. Wtedy istnieje pewność, że wpisy w listach danych są aktualne i każdą przybyłą krotkę z wejścia pierwszego łączy się z krotką z wejścia drugiego w momencie dopasowania.

Do komponentów niezależnych od danych możemy zaliczyć:

Komponent *Filtracji* weryfikuje krotki pod względem spełniania określonego warunku filtracji. Tutaj sprowadza się to do sprawdzenia spełnienia warunku filtracji i przesłania dalej krotki, jeśli spełnia lub odrzucenia krotki, gdy nie spełnia sprecyzowanego dla niej warunku.

Komponent *Unii* określa poziome złączenie krotek z N wejść. W przypadku tego komponentu pobieramy sekwencyjnie dane z kolejnych wejść i wysyłamy w kolejności przybycia.

Komponent *Projekcji* umożliwia wydzielenie podrelacji z relacji wyjściowej. W przypadku tego komponentu odbieramy sekwencyjnie dane z wejścia i na każdej otrzymanej krotce wykonujemy projekcję: wydzielenie wybranych kolumn, zgodnie z konfiguracją komponentu.

Komponent *Scalania* umożliwia poziome złączenie danych z kilku wejść. Tutaj pojawia się problem znaczników oraz różnych liczności danych na wejściach. Kolejność złączenia w przypadku scalania jest określona przez kolejność nadchodzenia danych na wejścia i nie jest deterministyczna (np. tak jak w przypadku złączenia, gdzie mamy równość klucza złączenia). Dlatego można skonfigurować komponent tak, aby łączył tylko nowe krotki (znacznik N) oraz zapisywał niedopasowane krotki pomiędzy kolejnymi etapami do późniejszego dopasowania.

Komponent *Generacji* pozwala generować dane na podstawie przybyłych danych. Dane mogą być generowane z określonym krokiem, począwszy od określonej wartości oraz mogą być wysyłane zamiast przybyłej krotki albo dodane na wskazane miejsce do tej krotki.

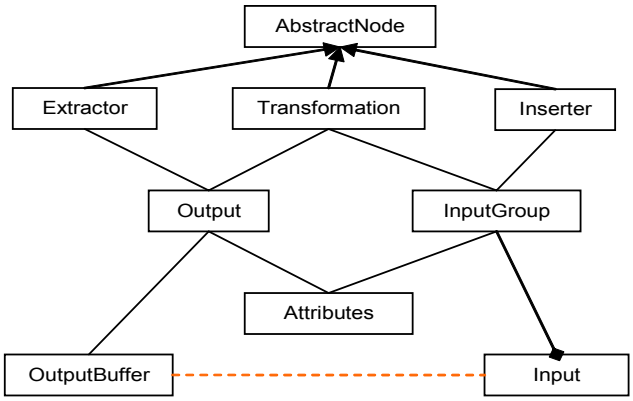
Komponent *Sortowania* uporządkowuje dane wg ustalonego porządku (rosnącym lub malejącym) na wskazanych kolumnach. Komponent jest blokujący, czyli na wyjściu pojawiają się krotki dopiero wtedy, gdy skończą się dane na wejściu.

Komponenty wstawiania danych.

Wyróżnia się inwertory do pliku tekstowego oraz inwertory do docelowej bazy danych. W przypadku komponentów wstawiania danych do docelowej tabeli w bazie danych, w zależności od typu przybyłej krotki musimy wykonać jedną z 3 operacji: INSERT dla krotek typu N, UPDATE dla krotek typu U oraz DELETE dla krotek typu D. W przypadku wstawiania danych do pliku problem jest większy, bowiem trudno w pliku zmienić konkretną linię. Dlatego w komponencie znajduje się mapa, która odzwierciedla klucz każdej krotki i jej położenie w pliku. Dzięki takiemu rozwiązaniu można odwołać się do każdej linii, aby móc ją zmienić (dla krotek U) lub usunąć (dla krotek typu D).

4. Implementacja systemu ETL(δ)

System ETL(δ) jest rozbudowanym o proces δ systemem ETL i opisany schematem głównych klas (rys. 4).

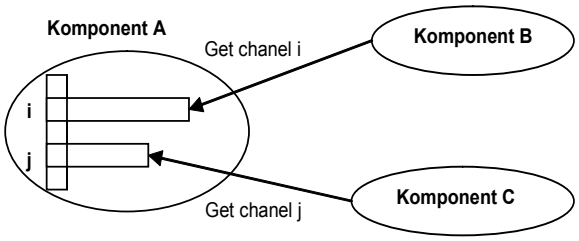


Rys. 4. Schemat głównych klas systemu
Fig. 4. Schema mains classes of system

Każdy komponent w systemie pochodzi od klasy bazowej ***AbstractNode*** i odpowiadającej mu jednej ze specjalizacji: ekstrakcji (klasa ***Extractor***), transformacji (klasa ***Transformation***) lub wstawiania (klasa ***Inserter***). Każda z tych klas posiada wejście (klasa ***InputGroup***) i/lub wyjście (klasa ***Output***) z/do komponentu. Na wejście komponentu składają się obiekty klasy ***Input***, które reprezentują pojedyncze połączenie pomiędzy sąsiadującymi komponentami. Na klasę wyjścia komponentu składa się bufor wyjściowy implementowany w klasie ***OutputBuffer***. Wyjście i wejście komponentu posiada określoną

relację w postaci obiektu klasy ***Attributes***. Wydzielenie obiektu klasy relacji pozwala na łatwe zarządzanie obiektami krotek przechodzących przez system, bowiem w takim przypadku pojedyncza krotka niesie tylko dane.

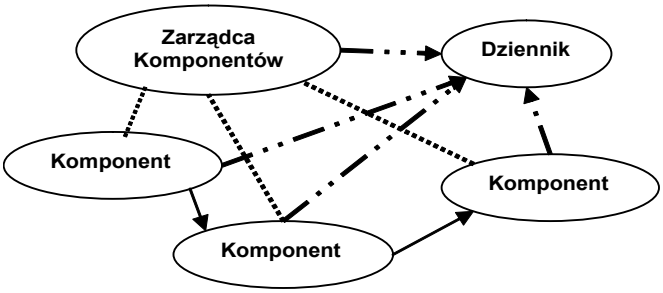
Połączenie pomiędzy sąsiednimi komponentami odbywa się na drodze obiektów: ***OutputBuffer*** komponentu poprzedzającego oraz ***Input[i]*** komponentu następującego (przerywana linia na rys. 4). Na wyjściu jest N kanałów odbiorczych (N jest liczbą następujących komponentów), do których z osobna wpływa kolejna kopia danych. Każdy następujący komponent posiada przydzielony mu numer kanału odbiorczego w buforze wyjściowym, co zapewnia niezależny odbiór danych (rys. 5).



Rys. 5. Przesył danych w systemie ETL(δ)
Fig. 5. Sending of data in the system ETL(δ)

W systemie można wyróżnić kilka typów połączeń (rys. 6):

- dane, będące połączeniem kierunkowym pomiędzy dwoma sąsiednimi komponentami, gdzie przekazywane są dane (pakiety krotek – linia ciągła),
- komunikaty, przekazywane na drodze: komponent a zarządca komponentów. Służą one sterowaniu działaniem systemu (linia wykropkowana),
- zapisy dziennika i błędy, przekazywane do osobnego obiektu dziennika, który je wszystkie magazynuje i przetwarza (linia przerywana).



Rys. 6. Rodzaje komunikatów w systemie ETL(δ)
Fig. 6. Kind of communication in the system ETL(δ)

Zarządca komponentów odpowiada za prawidłową pracę i konfigurację komponentów. W przypadku wystąpienia *wyjątku* kończy działanie systemu ETL(δ).

Konfiguracja systemu opiera się na przygotowanym pliku tekstowym z zapisaną listą komponentów i ich konfiguracji. Każdy komponent ma nadany unikalny identyfikator.

W pliku, poprzez określenie identyfikatorów komponentów źródłowych poprzedzających dany komponent, zaszyta jest także sieć połączeń grafu ETL(δ). Plik konfiguracji podzielony jest na wydzielone sekcje, gdzie każda sekcja odpowiada pojedynczemu komponentowi, a treść danej sekcji jest konfiguracją tego komponentu. Dodatkowo, w każdym pliku konfiguracyjnym znajduje się sekcja *System*, która odpowiada za wspólną konfigurację dla wszystkich komponentów.

5. Podsumowanie i wnioski

System ETL(δ) realizuje proces ekstrakcji danych, wykorzystujący ideę ciągłej integracji danych. Jest uzupełnieniem klasycznego systemu ETL o mechanizm procesu przyrostowej ekstrakcji danych w ramach detekcji zmian w danych źródłowych. System ETL(δ) gwarantuje pełną dostępność danych w hurtowni danych oraz zachowanie historii zmian danych. Wyniki badań potwierdzają fakt, iż uzyskana funkcjonalność przewyższa nakłady poniesione na detekcję zmian w danych oraz operowania na ich podzbiorze [15].

LITERATURA

1. Bruckner R., List B., Schiefer J.: Striving towards Near Real-Time Data Integration for Data Warehouses. DaWaK 2002, pp. 317-32.
2. Raden N.: Real time: get Real. Take the idea of a real-time data warehouse with a grain of salt, then realize the possibilities. Intelligent Enterprise, vol. 6, no.10, 2003.
3. Gorawski M., Jabłoński P.: Uniwersalne środowisko graficzne do modelowania procesów ekstrakcji i odtwarzania. Studia Informatica, vol. 26, 3(64), s. 7-28, 2005.
4. Gorawski M., Marks P.: Grouping and Joining Transformations in Data Extraction Process. XXI Autumn Meeting of the Polish Information Processing Society, Conference Proceedings, Wisła, Poland, ISBN 83-922646-0-6, PIPS pp. 105-113, 2005.
5. Gorawski M., Piekarek M.: Rozproszony proces ekstrakcji danych z protokołem SimpleRMI. Bazy Danych – Modele, Technologie, Narzędzia, Eds. S. Kozielski, WKiŁ, ISBN 83-206-1572-0, s. 43-50, 2005.
6. Gorawski M., Marks P.: Data Loading based on UB-Tree Index Implemented in Design-Resume/JavaBeans Environment. Studia Informatica, vol. 25, 1(57), pp. 141-154, 2004.
7. Gorawski M., Piekarek M.: Rozwojowe środowisko ETL/JavaBeans wzbogacone o rozproszone sortowanie danych. Praca zbiorowa „Współczesne problemy sieci komputerowych” WNT, s. 173-180, 2004.

8. Gorawski M.: Ekstrakcja i integracja danych w czasie rzeczywistym. Praca zbiorowa. „Współczesne problemy systemów czasu rzeczywistego", WNT, s. 435- 445, 2004.
9. Gorawski M.: 3 perspektywy procesu ekstrakcji danych. Praca zbiorowa „Strategie informatyzacji i zarządzanie wiedzą", Eds. Szyjewski Z., Nowak J., Grabara J.,WNT, ISBN-83-2004-3014-3, s. 295-341, 2004.
10. Gorawski M.: Charakterystyka procesu ekstrakcji danych. Studia Informatica, vol. 24, 4(56), s. 212-232, 2003.
11. Gorawski M., Piekarek M.: Rozwojowe środowisko ETL/Java Beans. Studia Informatica, vol. 24, 4(56), s. 288-302, 2003.
12. Gorawski M., Siódemak P.: Graficzne projektowanie aplikacji ETL. Studia Informatica, vol. 24, 4(56), s. 345-367, 2003.
13. Gorawski M.: Modelowanie procesu ekstrakcji danych. IV Konferencja Metody i systemy komputerowe w badaniach naukowych i projektowaniu inżynierskim, 26-28 listopada, 2003, Kraków, Poland, ONT, Ed. Tadeusiewicz R., Ligęza A., Szymkat M., ISBN 83-916420-1-1, s. 165-170, 2003.
14. Rosana L. de B. A. Rocha, Cardoso L., Souza J.: An Improved Approach in Data Warehousing ETLM Process for Detection of Changes in Source Data. SBBD 2003, pp. 253-266.
15. Gorawski M., Ciepluch M.: Ocena wydajności systemu przyrostowej ekstrakcji danych ETL(δ). II Konferencja Bazy Danych – Modele, Technologie, Narzędzia. (BDAS 2006).

Recenzent: Dr inż. Arkadiusz Sochan

Wpłynęło do Redakcji 19 stycznia 2006 r.

Abstract

ETL process (data extraction) in practice can take even up to 70% of the overall time destined for data warehouse realization. In goal of increasing data availability for data analysis in data warehouses authors propose adding changes detection mechanism to existing ETL process, and operation only on changed data. Such solution system ETL(δ) enables changes history archivisation in starting data, and actualization may have place along with access to data for analysis.

Adresy

Marcin GORAWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, Marcin.Gorawski@polsl.pl.
Mariusz CIEPLUCH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, m.ciepluch@polsl.pl