STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Dariusz R. AUGUSTYN, Piotr BAJERSKI, Robert BRZESKI
Politechnika Śląska, Instytut Informatyki

ZACHOWANIE SPÓJNOŚCI DANYCH W WYBRANYCH SYSTEMACH NOSQL

Streszczenie. Zachowanie spójności przetwarzanych danych jest problemem ogólnym, w szczególności dotyczącym systemów NoSQL, w których efektywność przetwarzania jest czynnikiem najistotniejszym. Wysoką efektywność przetwarzania w bazach NoSQL osiąga się przez uproszczenie mechanizmów wewnętrznych, w tym, częściowo, kosztem utrudnienia programowania – aplikacje korzystające z baz NoSQL muszą zawierać kod odpowiedzialny za utrzymanie spójności danych. Programowe metody zachowania spójności przetwarzanych danych rozważono w kontekście wybranych systemów NoSQL: MongoDB, Mircosoft Windows Azure, Apache Cassandra, Oracle NoSQL.

Słowa kluczowe: bazy danych NoSQL, spójność, współbieżność, transakcyjne przetwarzanie, MongoDB, Mircosoft Windows Azure, Apache Cassandra, Oracle NoSQL

DATA CONSISTENCY PRESERVATION IN SELECTED NOSQL SYSTEMS

Summary. Consistency preservation is a general problem in data processing, especially in NoSQL systems. NoSQL databases introduce a new approach, in which the efficiency is a major feature. It is partially achieved at the expense of some difficulties of programming. The application should handle situations which may introduce inconsistency in data. The paper presents the methods of data consistency preservation in selected NoSQL: MongoDB, Mircosoft Windows Azure, Apache Cassandra, Oracle NoSQL.

Keywords: NoSQL databases, consistency, concurrency, transaction processing, MongoDB, Mircosoft Windows Azure, Apache Cassandra, Oracle NoSQL

1. Wstęp

Bazy danych NoSQL wyznaczają kolejny etap w rozwoju bazodanowych systemów informatycznych. Uproszczony w stosunku do relacyjnych systemów zarządzania bazami danych (RSZBD) model danych pozwala na tworzenie rozwiązań tylko dla specyficznych zastosowań, ale za to charakteryzujących się szczególnie dużą efektywnością.

Wśród wielu koncepcji leżących u podstaw NoSQL znajduje się idea baz danych typu klucz-wartość. Sama koncepcja jest oczywista i powszechnie znana, natomiast implementacje systemów zarządzania klasy NoSQL rozszerzają ją w specyficzny dla nich sposób.

W artykule dokonano przeglądu wybranych systemów NoSQL. Za pomocą przykładów zaprezentowano sposoby manipulowania danymi, udostępniane w konkretnych systemach NoSQL (bazach dokumentowych lub opartych na koncepcji klucz-wartość).

Niezależnie od uproszczeń, jakie niesie ze sobą podejście NoSQL, bez jakiegokolwiek wsparcia rozwiązania problemu zachowania/kontroli spójności modyfikowanych danych oraz atomowości przetwarzania (chociażby w najbardziej podstawowym znaczeniu), bazy NoSQL nie mogłyby znaleźć szerszego praktycznego zastosowania (poza systemami udostępniającymi dane tylko do odczytu).

Głównym celem tego przeglądowego artykułu jest omówienie specyficznych cech wybranych systemów NoSQL w kontekście poprawności przetwarzania. W szczególności chodzi tu o wskazanie występowania braków pewnych mechanizmów programowych znanych z RSZBD, zamierzonych przez twórców. Artykuł może stanowić zbiór wskazówek w przedstawionym zakresie dla programistów systemów bazodanowych.

2. Mongo DB

MongoDB jest jedną z najpopularniejszych baz NoSQL w kategorii baz dokumentowych. Dostępna jest na zasadach otwartego oprogramowania (ang. *open source*), na licencji GNU AGPL v 3.0 [2, 3]. Opis zamieszczony w niniejszym rozdziale dotyczy MongoDB w wersji 2.0.2.

MongoDB może być uruchomiona jako pojedyncza instancja serwera dokumentowej bazy danych. Może także zostać skonfigurowana do pracy w systemie rozproszonym na wielu węzłach. Jest to tzw. tryb *sharding*, umożliwiający skalowanie poziome bazy MongoDB. Pojedyncza instancja, oparta na procesie *mongod*, może być skonfigurowana do wersji rozproszonej z dostępem przez kontroler *mongos*. Wersja rozproszona może zawierać do 1000 węzłów. Przy skalowaniu poziomym, dane są podzielone na wiele serwerów, natomiast z zewnątrz, za

pośrednictwem kontrolera mongos, widzimy je jako jeden serwer. W ramach poszczególnych części dane mogą być także zreplikowane w trybie *master-slave*, czyli zapis następuje do węzła master, a odczyt jest możliwy z dowolnego węzła. W takim środowisku, dzięki użyciu kontrolera *mongos*, wiele operacji może być wykonywanych równolegle. Są one jednak wykonywane niezależnie od siebie – bez blokowania innych operacji. Otrzymujemy dzięki temu dużą wydajność serwera, tzn. szybkość przetwarzania danych. Natomiast w systemie opartym na pojedynczej instancji *mongod* operacje mogą być blokowane. Możliwa jest dowolna liczba współbieżnych operacji odczytu danych, natomiast operacja zapisu blokuje wszystkie inne operacje.

2.1. Operacje Atomowe

MongoDB wspiera operacje atomowe na pojedynczych dokumentach. Przy czym nie ma tutaj klasycznego blokowania ani obsługi złożonych transakcji z kilku powodów [2, 3]:

1. W środowisku rozproszonym blokowanie może spowodować znaczące spowolnienie działania systemu. Podstawowym celem stosowania tej bazy jest szybkość przetwarzania i dostępu do danych.
2. Celem uproszczenia sposobu blokowania danych jest uniknięcie niepożądanych sytuacji typu zakleszczenia (blokady wzajemne), które m.in. negatywnie wpływają na wydajność.
3. Twórcy MongoDB nie chcą, aby wykonywanie operacji blokujących znaczny zakres danych przez długi czas uniemożliwiało realizacje innych operacji (w szczególności realizację prostych zapytań).

Instrukcja modyfikacji „update” umożliwia atomowe zastąpienie dokumentu spełniającego zadane kryteria przez podany dokument. W celu atomowej modyfikacji wybranych pól pojedynczego dokumentu MongoDB udostępnia kilka modyfikatorów [1]. Są to:

$set – zapisanie konkretnej wartości w polu,
$unset – usunięcie z dokumentu podanego pola,
$inc – zwiększenie wartości pola o podaną wartość,
$push – dodanie podanej wartości do pola będącego tablicą,
$pushAll – dodanie kilku wartości do pola będącego tablicą,
$pull – usunięcie podanej wartości z pola będącego tablicą,
$pullAll – usunięcie kilku wartości z pola będącego tablicą,
$bit – operacje na poziomie bitowym.

Przykładem wykorzystania wymienionych instrukcji może być wywołanie:

```
db.kolekcja.update({klucz1 : 1},{ $set : {klucz2 : 3 },$inc:{klucz3 : 4}}, true,
true)
```

dotyczące aktualizacji dokumentów, w którym pole o nazwie „klucz1" ma wartość 1 i gdzie następuje: jednoczesne ustawienie pola „klucz2" na 3 (lub dodanie, jeśli nie było pola o takiej nazwie w dokumencie) oraz zwiększenie wartości pola „klucz3" o wartość 4.

Można użyć trybu „multi-update", aby dokonać operacji modyfikacji na wielu obiektach – dokumentach. Standardowo „multi-update" pozwala innym operacjom (zapisującym dane) na pracę współbieżną. Jeśli chcemy, aby przy modyfikowaniu danych dla jakiegoś zakresu dokumentów taka zmiana była całkowicie izolowana (tzn. aby inne, współbieżne procesy zapisu nie były realizowane w czasie modyfikacji danego zakresu dokumentów), można użyć flagi '$atomic' [1]:
brak izolacji:

```
db.kolekcja.update( { klucz1 : 1 } , { $inc : { klucz2 : 4 } } , false , true );
```

operacja izolowana:

```
db.kolekcja.update( { klucz1 : 1 , $atomic : true } , { $inc : { klucz2: 4 } } ,
false , true );
```

przy czym ciąg modyfikacji nie jest tutaj atomowy – nie mamy pewności, że zostaną wykonane wszystkie operacje albo żadna z nich. Słowo kluczowe $atomic w zasadzie oznacza poziom izolacji, tzn. że jest to jedyny proces zapisujący podczas realizacji modyfikacji, czyli każda modyfikacja jest realizowana bez ingerencji ze strony innej [1].

Flaga $atomic nie działa dla systemu rozproszonego – „sharding". Jeżeli zostanie użyta, to modyfikacja nie zostanie zrealizowana.

W trakcie operacji usuwania możliwa jest sytuacja, w której inne operacje wykonywane współbieżnie będą mogły mieć dostąp do usuwanych danych. Jeżeli jest to sytuacja niepożądana, to możemy wyizolować operację usuwania przez dodanie flagi '$atomic' [1]:

```
db.kolekcja.remove( { ocena: { $lt : 4.0 }, $atomic : true } )
```

Podczas wykonywania tej operacji inne, potencjalnie współbieżne operacje są blokowane. Aby użyć takiej opcji, kolekcja, na której realizujemy operacje, nie może być rozproszona.

W związku z ograniczeniem atomowości jedynie do pojedynczych operacji wykonywanych na jednym dokumencie zaistniała potrzeba opracowania rozwiązań obsługujących sytuacje bardziej złożone, które również wymagają zachowania spójności i atomowości. W dalszej części zaprezentowano kilka przykładów w tym zakresie.

2.2. Zmodyfikuj, jeśli aktualne (ang. *update if current*)

Pewną strategią wykonania zestawu operacji z zachowaniem spójności jest modyfikowanie danych tylko wtedy, gdy nie zostały zmienione w międzyczasie (od ostatniego pobrania). Wykonujemy to następująco [1]:

1. Pobieramy obiekt.
2. Modyfikujemy go lokalnie.
3. Wysyłamy żądanie modyfikacji do nowej wartości, jeśli „w międzyczasie stara" wartość nie uległa zmianie.

Jeśli operacja się nie powiedzie, zaczynamy jeszcze raz od początku. Na przykład załóżmy, że chcemy pobrać towar z magazynu. Po jego pobraniu musimy odpowiednio pomniejszyć jego ilość w magazynie. Jeżeli mamy np.:

```
db.magazyn.save({"towar" : "nazwaTowaru" , "ilosc" : 10});
```

to następujący kod realizuje to zadanie [1]:

```
kolekcja=db.magazyn     // dodatkowa zmienna - 'kolekcja'
dok = kolekcja.findOne({towar:'nazwaTowaru'})  // pobieramy dokument i przypisu-
jemy do   zmiennej 'dok'
{
        "_id" : ObjectId("4f22c96d6a2ef7d3de7a0244"),
        "towar" : "nazwaTowaru",
        "ilosc" : 10
}  // wynik pobrania
ilosc_poprzednia = dok.ilosc ;    // wartość klucza 'ilosc' przypisujemy do
zmiennej 'ilosc_poprzednia' - 10
--dok.ilosc ;// zmniejszamy o 1 wartość klucza 'ilosc' dla zmiennej 'dok' prze-
chowującej pobrany dokument - 9
kolekcja.update({_id:dok._id, ilosc:ilosc_poprzednia}, dok);  // modyfikujemy
dokument o identyfikatorze 'dok._id' czyli wcześniej znaleziony, jeżeli wartość
'ilosc' jest równa starej wartości przetrzymywanej w zmiennej 'ilosc_poprzednia'
(jeżeli ten warunek nie zostanie spełniony to modyfikacja nie zostanie wykona-
na);  dok - dokument 'dok' z nową wartością.
dok = kolekcja.findOne({towar:'nazwaTowaru'})
{
        "_id" : ObjectId("4f22c96d6a2ef7d3de7a0244"),
        "towar" : "nazwaTowaru",
        "ilosc" : 9
}
```

2.3. Zatwierdzenie dwufazowe (ang. *two-phase commit*)

MongoDB zapewnia atomowość dla operacji wykonywanych na pojedynczym dokumencie. Oczywiście w rzeczywistych zastosowaniach mogą występować sytuacje, w których występują transakcje obejmujące kilka dokumentów. Implementacja obsługi takich transakcji może być oparta na metodzie zatwierdzenia dwufazowego (ang. *two-phase commit*) [4].

Znanym przykładem transakcji jest przelew środków z jednego konta na drugie, z K1 na K2 w wiarygodny sposób. W systemie zarządzania relacyjną bazą danych środki są odejmowane z konta K1 i dodawane do konta K2 w jednej transakcji, co zapewnia nam atomowość całej operacji. W Mongo DB rozwiązanie opiera się na wykorzystaniu „two-phase commit" – zatwierdzania dwufazowego [1, 4].

Dalej przedstawiono przykład [1, 4] wykorzystania mechanizmu zatwierdzania dwufazowego dla kolekcji przechowującej odpowiednie konta (w postaci dokumentów). W dokumen-

cie tym mamy m.in. klucz „wykonywaneTransakcje: []" – z pustą wartością; w trakcie wykonywania danej transakcji będzie on przechowywał jej numer _id:

```
db.konto.save({nazwa: "K1", saldo: 100, wykonywaneTransakcje: []}) //konto K1
db.konto.save({nazwa: "K2", saldo: 100, wykonywaneTransakcje: []}) //konto K2
db.konto.find()
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff1"), "nazwa" : "K1", "saldo" : 100,
"wykonywaneTransakcje" : [ ] }
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff2"), "nazwa" : "K2", "saldo" : 100,
"wykonywaneTransakcje" : [ ] }
```

Potrzebujemy drugiej kolekcji reprezentującej transakcje (ze stanem „doWykonania"):

```
db.opisTransakcji.save({zrodlo: "K1", cel: "K2", kwota: 10, stan: "doWykona-
nia"})
db.opisTransakcji.find()
{ "_id" : ObjectId("4f2a7a5990dd60392599bff3"), "zrodlo" : "K1", "cel" : "K2",
"kwota" : 10, "stan" : "doWykonania" }
```

Opis przebiegu transakcji wraz z uzyskiwanymi wynikami poszczególnych operacji [1, 4]:

W **kroku 1** ustawiamy stan transakcji na „wTrakcie":

```
t = db.opisTransakcji.findOne({stan: "doWykonania"}) //"pobranie" transakcji i
przypisanie jej do zmiennej 't'.
{
        "_id" : ObjectId("4f2a7a5990dd60392599bff3"),
        "zrodlo" : "K1",
        "cel" : "K2",
        "kwota" : 10,
        "stan" : "doWykonania"
}
db.opisTransakcji.update({_id: t._id}, {$set: {stan: "wTrakcie"}}) // modyfiku-
jemy stan na 'wTrakcie'
db.opisTransakcji.find()
{ "_id" : ObjectId("4f2a7a5990dd60392599bff3"), "zrodlo" : "K1", "cel" : "K2",
"kwota" : 10, "stan" : "wTrakcie" }
```

W **kroku 2** realizujemy operacje transakcji (zmiana salda konta) na obu kontach, upewniając się, że transakcja nie jest już w trakcie realizacji, czyli sprawdzamy klucz „wykonywaneTransakcje" i wprowadzamy jego zmianę, wpisując _id przetwarzanej transakcji:

```
db.konto.update({nazwa: t.zrodlo, wykonywaneTransakcje: {$ne: t._id}}, {$inc:
{saldo: -t.kwota}, $push: {wykonywaneTransakcje: t._id}})
db.konto.update({nazwa: t.cel, wykonywaneTransakcje: {$ne: t._id}},
{$inc:{saldo: t.kwota}, $push: {wykonywaneTransakcje: t._id}})
db.konto.find()
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff1"), "nazwa" : "K1", "saldo" : 90,
"wykonywaneTransakcje" : [ ObjectId("4f2a7a5990dd60392599bff3") ] }
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff2"), "nazwa" : "K2", "saldo" : 110,
"wykonywaneTransakcje" : [ ObjectId("4f2a7a5990dd60392599bff3") ] }
```

W **kroku 3** ustawiamy stan transakcji na „zatwierdzona":

```
db.opisTransakcji.update({_id: t._id}, {$set: {stan: "zatwierdzona"}})
db.opisTransakcji.find()
{ "_id" : ObjectId("4f2a7a5990dd60392599bff3"), "cel" : "K2", "kwota" : 10,
"stan" : "zatwierdzona", "zrodlo" : "K1" }
```

W kroku 4 usuwamy z dokumentów opisujących konta wpis _id zrealizowanej transakcji:

```
db.konto.update({nazwa: t.zrodlo}, {$pull: {wykonywaneTransakcje: Objec-
tId("4f2a7a5990dd60392599bff3")}})
db.konto.update({nazwa: t.cel}, {$pull: {wykonywaneTransakcje: Objec-
tId("4f2a7a5990dd60392599bff3")}})
db.konto.find()
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff1"), "nazwa" : "K1", "saldo" : 90,
"wykonywaneTransakcje" : [ ] }
{ "_id" : ObjectId("4f2a7a3a90dd60392599bff2"), "nazwa" : "K2", "saldo" : 110,
"wykonywaneTransakcje" : [ ] }
```

W kroku 5 ustawiamy stan transakcji na „zrealizowana”:

```
db.opisTransakcji.update({_id: t._id}, {$set: {stan: "zrealizowana"}})
db.opisTransakcji.find()
{ "_id" : ObjectId("4f2a7a5990dd60392599bff3"), "cel" : "K2", "kwota" : 10,
"stan" : "zrealizowana", "zrodlo" : "K1" }
```

Jest to oczywiście uproszczona implementacja metody przelewu środków z wykorzystaniem mechanizmu zatwierdzania dwufazowego.

3. Windows Azure Table

3.1. Windows Azure Storage – modele utrwalania i pobierania danych

Windows Azure Storage [6, 9] to zestaw mechanizmów programowych, pozwalający aplikacjom przeznaczonym do przetwarzania w chmurze na efektywne utrwalanie, modyfikowanie i pobieranie danych. W ramach Windows Azure Storage dostępne są następujące mechanizmy/usługi przechowywania danych:

- Windows Azure Queue – pozwalający na synchronizację procesów przetwarzających przez dane, funkcjonalnie zbliżony do kolejki FIFO,
- Windows Azure Blob – umożliwiający obsługę i przechowywanie dużych danych (tzw. danych binarnych),
- Windows Azure Table – pozwalający na obsługę i przechowywanie danych strukturalnych – podejście zgodne modelem danych typu klucz-wartość,
- Windows SQL Azure – zapewniający funkcjonalność serwera relacyjnej bazy danych w chmurze (praktycznie udostępnienie niemal wszystkich usług SZBD Microsoft SQL Server).

3.2. Windows Azure Table – obsługa danych typu klucz-wartość

W ramach Windows Azure Table [7, 8, 9] można tworzyć tablice (ang. *table*), które stanowią zbiór encji (ang. *entity*), czyli „wierszy tablic”. Wiersze składają się z właściwości

(ang. *property*) różnych typów prostych (*Binary, Bool, DateTime, Double, GUID, Int, Int64, String*). W dużym uproszczeniu własności można traktować jako odpowiednik kolumn w relacyjnej bazie danych. Tablica może przechowywać encje o różnej strukturze (struktura tablicy nie jest „sztywna"). Istnienie trzech właściwości: *PartitionKey, RowKey* i *TimeStamp* jest obowiązkowe. Klucz każdego wiersza składa się z dwu elementów: *PartitionKey* i *RowKey*. Dane z tą samą wartością *PartitionKey* należą do tzw. partycji. *PartitionKey* jest wykorzystywany przez wewnętrzne mechanizmy *Windows Azure* w celu zapewnienia wydajności aplikacji przetwarzającej (skalowalności) przez możliwość automatycznego rozproszenia przetwarzania na wiele węzłów z dokładnością do partycji. *RowKey* pozwala na jednoznaczną identyfikację w obrębie partycji. Teoretycznie nie ma ograniczeń liczby tablic, liczby własności w encjach i liczby encji w tablicy.

Zgodnie z paradygmatem modelu danych klucz-wartość wyszukiwanie danych i sortowanie według klucza powinno być wykonywane szybko. Natomiast sortowanie i wyszukiwanie po atrybutach niekluczowych, chociaż możliwe, nie będzie efektywne. Stąd szczególnie istotną decyzją projektową, wpływającą na wydajność rozwiązania, jest decyzja o budowie klucza, tzn. jakie dane, w sensie merytorycznym, będą umieszczane w ramach składowych klucza encji *PartitionKey* i *RowKey*. „Rozszerzanie zawartości" klucza o dodatkowe dane (zależne od klucza merytorycznego) prowadzi do złamania zasady, stwierdzającej, że schemat tablicy ma spełniać postacie normalne pierwszą (1NF) i trzecią (3NF) (tzn. dane „w kluczu" encji nie są już atomowe i zawierają elementy zależne od faktycznego klucza merytorycznego). Oczywiście takie podejście jest zamierzone, tzn. ma prowadzić do poprawy efektywności wyszukiwania kosztem odejścia od koncepcji normalizacji, znanych z podejścia relacyjnego. Usługi Windows Azure Table mają znaleźć zastosowanie w systemach wymagających specyficznego przetwarzania – „pomiędzy" przetwarzaniem danych niestrukturalnych (obsługiwanym przez Windows Azure Blob) a przetwarzaniem z użyciem modelu relacyjnego (obsługiwanym przez Windows SQL Azure).

Przykłady ilustrujące użycie Windows Azure Table wykonano z wykorzystaniem oprogramowania Windows Azure SDK w wersji 1.6.

3.3. Programowanie z wykorzystaniem Windows Azure Table

Programiści, w celu uzyskania dostępu do *Azure Table*, mogą zaprogramować obsługę protokołu *http RESTful web services* [11, 12] albo – łatwiej – bezpośrednio skorzystać z *ADO.NET Data Services API*. Zestawianie operacji typu CRUD [8] na encjach *Azure Table* dla *RESTfull* (komenda *HTTP*) i dla *ADO.NET Data Services* zostało przedstawionie w tabeli 1.

Tabela 1

Wielkości czcionek i atrybuty elementów makiety artykułu

Operacja	**ADO.NET Data Services**	**Komenda HTTP**
Zapytanie **do pojedynczej** tablicy	LINQ Query	GET
Aktualizacja całej encji	UpdateObject() & SaveChanges(`SaveChangesOptions.ReplaceOnUpdate`)	PUT
Aktualizacja częściowa encji	UpdateObject() & SaveChanges()	MERGE
Utworzenie encji	AddObject() & SaveChanges()	POST
Skasowanie encji	DeleteObject() & SaveChanges()	DELETE

Sposób wykorzystania funkcjonalności ADO NET. Data Services zilustrowano za pomocą fragmentów prostych programów w języku C#, które są odpowiedzialne za:

- stworzenie tablicy *Faktury*,
- wstawienie nowej encji – obiektu klasy *Faktura* – do tablicy *Faktury*,
- pobieranie encji z tablicy *Faktura*, zgodnie z zadanymi kryteriami wyszukiwania (zapytanie LINQ).

Tablica *Faktury* jest tworzona przez wywołanie:

```
var storageAccount = CloudStorageAccount.FromConfigurationSetting (…);
var cloudTblCli = storageAccount.CreateCloudTableClient();
cloudTblCli.CreateTableIfNotExist("Faktury");
```

Klasa *Faktura* (określająca postać encji) została zdefiniowana następująco:

```
class Faktura : TableServiceEntity {
  public Faktura() {}
  public Faktura (Int64 idFaktura) {
    DataWystawienia = DateTime.Now;
    this.PartitionKey = DataWystawienia.Year.ToString();
    this.RowKey = (DataWystawienia.Month.ToString()).PadLeft(2,'0')
                  + "-" + idFaktury;
    this.IdFaktury = idFaktury;
  }
  public Int64 IdFaktury { get; set; }
  // Tekstowy identyfikator Klient(np. „FirmaOdeonSA01”)
  public string NazwaKlienta { get; set; }
  public string Opis { get; set; }
  public int Status { get; set; }  // 1 – zapłacone, 0 – niezapłacone
  public DateTime DataWystawienia { get; set; }
 }
```

Wszystkie publiczne własności klasy (z tzw. *getterami* i *setterami*) stają się własnościami encji *Faktura* po umieszczaniu w tablicy *Faktury*.

W klasie *Faktura* klucz jest zbudowany z trzech wartości typu *String*, tj.: rok, miesiąc, *IdFaktury* (gdzie rok i miesiąc dotyczą daty wystawienia faktury). Przyjęto, że *PartitionKey* jest rok (np. *PartitionKey* == ”2012”), a *RowKey* to złożenie numeru miesiąca, znaku ‘-‘ i identyfikatora (np. ”03-3203” dla faktury o identyfikatorze 3203, wystawionej w marcu).

Wstawienie encji z identyfikatorem 3201 do tablicy *Faktury* (w ramach konta <account>) można zrealizować następująco:

```
Int64 id = 3201;
Faktura fk = new Faktura (id) {
  NazwaKlienta = "OrkanSA",
  Opis = "Faktura za produkt500",
  Status = 1
};
serviceUri = new Uri("http://<account>.table.core.windows.net");
var context = new DataServiceContext(serviceUri);
context.AddObject("Faktury", fk);
DataServiceResponse response = context.SaveChanges();
```

Pobranie wszystkich faktur ze stycznia 2012 roku odbywa się przez wywołanie:

```
serviceUri = …
var context = …
var serviceUri = new Uri("http://<account>.table.core.windows.net");
DataServiceContext context = new DataServiceContext(serviceUri);

var faktury =
  from fk in context.CreateQuery<Faktura>("Faktury")
  where fk.PartitionKey == "2012"
    && fk.RowKey.CompareTo("01") >= 0
    && fk.RowKey.CompareTo("02") < 0
  select fk;

foreach (var fk  in faktury) {
// przetwarzanie faktur ze stycznia 2012r
}
```

Na uwagę zasługuje fakt, że sformułowanie przedstawionego zapytania powinno gwarantować jego efektywną realizację, ponieważ kryterium wyszukiwania jest zdefiniowane tylko na wartościach klucza, tzn. na całym *PartitionKey* i początku *RowKey*.

Pokazane przykłady miały wskazać zaletę podejścia zastosowanego w Windows Azure Table, tzn. przejrzysty model programistyczny (LINQ-based API), pozwalający na przetwarzanie danych zbliżone do dobrze znanego modelu dostępu do danych – „LINQ to SQL" czy „LINQ to Object", co w społeczności wytwórców oprogramowania jest postrzegane jako zaleta i dodatkowo powinno przyczynić się do wzrostu zainteresowania tym rozwiązaniem.

3.4. Problem współbieżnych aktualizacji

Typowy scenariusz, związany ze współbieżną aktualizacją pojedynczej encji przez równocześnie działające procesy zostanie przedstawiony przez omówienie kroków zamieszczonego dalej scenariusza działań [8].

Wersja każdej encji jest przechowywana po stronie systemu Azure. W momencie pobrania encji system Azure dostarcza wersję encji do procesu klienta (w ramach *HTTP ETag*). Po aktualizacji encji po stronie klienta do systemu Azure dostarczane są zmodyfikowane własności encji oraz wersja (*ETag* w ramach nagłówka *If-Match* wysyłanego żądania). Jeśli encja nie była zmieniana (zgodność wersji podesłanej z istniejącą po stronie systemu Azure), to dane są

zaakceptowane, a nowa wersja encji jest odsyłana do klienta. Jeśli dane były zmienione, to żądanie jest odrzucone i błąd http 412 – „precondition failed" – zostaje odesłany do klienta. Zazwyczaj, dla obsługi takiej sytuacji, program klienta będzie napisany w ten sposób, że oryginalna encja zostanie pobrana ponownie (wraz z oznaczeniem aktualnej wersji), zaktualizowana lokalnie i odesłana do systemu Azure.

Obsługa sytuacji w opisany sposób wymaga odpowiedniego ustawienia własności kontekstu – tzn. *context.MergeOption = MergeOption.PreserveChanges* (domyślna wartość to *MergeOption.AppendOnly*). Sposób rozwiązania problemu współbieżnej aktualizacji (obsługa tzw. sytuacji *lost update*) ilustruje następujący kod źródłowy:

```
context.MergeOption = MergeOption.PreserveChanges;
var faktury =
  from fk in context.CreateQuery<Faktura>("Faktury")
  where fk.PartitionKey == "2012"
    && fk.RowKey == "01-3150"
  select fk
var myfk = faktury.FirstOrDefault();
if (myfk == null) return; // sprawdzenie czy brak faktury 2012,01-3150
myfk.Status = 1 ;
myfk.Opis = ”Wreszcie zapłacona” ;
  try {
    context.UpdateObject(myfk);
    DataServiceResponse response = context.SaveChanges();
  } catch (DataServiceRequestException e) {
      OperationResponse response = e.Response.First();
      if (response.StatusCode == (int)HttpStatusCode.PreconditionFaile {
        // ponowne pobranie encji z systemu Azure i próba aktualizacji
   }
  }
```

Możliwa jest (chociaż na ogół niezalecana) praca w trybie bezwarunkowej aktualizacji (nadpisanie danych bez kontroli wersji), tzn. ustawienie „dowolnej" wersji nadpisywanej encji, czyli ustawienie zawartości *If-Match hedera* żądania na „*":

```
context.Detach(myfk);
// aktualizacja pól obiektu myfk
context.AttachTo("Faktura", myfk, "*");
context.UpdateObject(myfk);
```

3.5. Przetwarzanie transakcyjne – transakcje grupowe

Ogólnie w Windows Azure Table nie obowiązuje model przetwarzania transakcyjnego typu ACID [5]. Jednak model ten funkcjonuje w odniesieniu do przetwarzania pojedynczej encji jednej tablicy, tzn. ACID na poziomie pojedynczej operacji wstawiania/aktualizacji/kasowania encji.

System Windows Azure Table zapewnia poziom izolacji transakcji – *Snapshot Isolation Level* [13], ale dotyczy on jedynie pojedynczych zapytań oraz danych zawartych w ramach tej samej partycji (jest to więc kolejne – oprócz wydajności – kryterium do uwzględnienia przy konstrukcji *PartitionKey*).

W ramach Windows Azure Table dodatkowo dostępny jest mechanizm grupowych transakcji [8, 10]. Jednak encje objęte tym mechanizmem muszą należeć do tej samej tablicy i tej samej partycji. Dodatkowo obowiązują jeszcze inne ograniczenia, np.: maksymalnie 100 encji, sumaryczna zajętość nie może przekraczać 4MB. Przykład zastosowania „grupowych transakcji" zilustrowano następującym kodem źródłowym:

```
for (int index = 0; index < 20; index++){
  Faktura fk  = new Faktura (
         // inicjalizacja pól ; obiekty z tym samym PartitionKey !!!
    );
  context.AddObject(fk);
}
// Wszystkie aktualizacje w ramach tzw. single batch request
DataServiceResponse response =
   context.SaveChanges(SaveChangesOptions.Batch);
```

O ile zagadnienie zachowania spójności przetwarzania w zakresie pojedynczej tablicy (ang. *single-table consistency problem*) jest zrealizowane przez system Azure, o tyle problem spójności przetwarzania danych w różnych tablicach pozostających z sobą w relacji (ang. *cross-table consistency problem*) nie jest systemowo rozwiązywany[1].

4. Apache Cassandra

Apache Cassandra jest popularną bazą typu klucz-wartość dostępną na zasadach otwartego oprogramowania (ang. *open source*) zgodnie z licencją Apache License 2.0. Model danych Cassandry opiera się na koncepcjach zastosowanych w Google BigTable, a technologie przetwarzania rozproszonego – na koncepcjach z Amazon Dynamo. Zamieszczony dalej opis rozwiązania dotyczy systemu Apache Cassandra w wersji 1.0.7.

System ten wykorzystuje model P2P (Peer-to-Peer) w przeciwieństwie do większości baz NoSQL, które wykorzystują model Master-Slave [14]. Użycie modelu P2P zapewnia bardzo dużą wydajność, jednak za cenę słabej spójności (ang. *weak consistency*). Podobnie jak Dynamo baza Cassandra charakteryzuje się własnością ostatecznej spójności (ang. *eventually consistent*). Implementacja P2P w systemie Cassandra opiera się na protokole bazującym na plotkach (ang. *gossip-base protocole*), odpowiedzialnym za dystrybucję informacji o stanie poszczególnych węzłów w klastrze oraz za balansowanie kluczy pomiędzy węzłami; do synchronizacji replikacji jest wykorzystywany mechanizm Anti-Entropy [18].

[1] W szczególnych wypadkach sugerowane jest dość uciążliwe pod względem kodowania, chociaż skuteczne, „obejście" problemu, w którym zleca się jakiemuś procesowi typu *WokerRole* „zadanie do wspólnej aktualizacji encji z kilku związanych ze sobą tablic" przez „kolejkę" (tzn. Windows Azure Queue). W obiekcie tym precyzuje się szczegóły „zadania" – identyfikatory kilku encji do modyfikacji. Korzysta się tu z własności, że nieudane wykonanie zadania, jeśli zostało przekazane do „wykonania przez kolejkę", będzie automatycznie wznawiane przez system (aż do skutku). Aby uniemożliwić działania na encjach objętych taką „transakcją", w ramach realizacji zadania, wstępnie, „jawnie blokuje się" te encje przez ustawienie stanu encji na „invalid" [8].

4.1. Model danych

Cassandra rozszerza koncepcję baz typu klucz-wartość o pojęcie rodzin kolumn (ang. *column family*), pozwalających grupować powiązane informacje [14, 17]. Rodziny kolumn można traktować jak pojemniki na kolumny i przechowywane w nich wartości, przy czym – w przeciwieństwie do klasycznych, relacyjnych baz danych – nie jest wykorzystywana koncepcja schematu. W rodzinie kolumn wiersz jest mapą odwzorowującą nazwy kolumn na wartości kolumn, tzn. odwołanie do konkretnej wartości następuje przez:

```
<nazwa_rodziny_kolumn>[<klucz_obiektu>][<nazwa_kolumny>],
```

gdzie nawiasy kwadratowe są częścią składni. Taka organizacja danych umożliwia każdemu wierszowi posiadanie innego zestawu kolumn. (Apache Cassandra dostarcza również bardziej złożonych konstrukcji wykorzystujących pojęcie superkolumn). Rodziny kolumn służą jedynie do organizowania przechowywania danych i nie udostępniają żadnego wsparcia dla obsługi współbieżnego dostępu.

Rodziny kolumn są umieszczane w przestrzeniach kluczy (ang. *keyspace*), w przybliżeniu odpowiadających schematom w klasycznych relacyjnych bazach danych. Replikacje są kontrolowane w ramach przestrzeni kluczy.

4.2. Operacje

System Cassandra udostępnia bardzo prosty zbiór operacji: zapisu (*put*), odczytu (*get* i *list*) oraz usuwania (*del*). Natomiast nie jest dostępna operacja aktualizacji. Operacja zapisu jest atomowa, ale nie izolowana. Dokładna semantyka operacji zależy od poziomu spójności (ang. *consistency level*) [19]:

1. Poziom ALL zapewnia dla operacjom zapisu, że nowa wartość zostanie zapisana we wszystkich replikach; w przypadku operacji odczytu wyszukiwanie odbywa się we wszystkich replikach i zwracana jest wartość z najświeższym znacznikiem czasowym. Brak dostępności którejkolwiek z replik uniemożliwia wykonanie operacji.
2. Poziomy QUORUM, EACH_QUORUM i LOCAL_QUORUM zapewniają wykonanie operacji na N/2 + 1 replikach, z tym że w pierwszym przypadku chodzi o wszystkie repliki w systemie, w drugim o repliki w każdym z centrów danych, a w ostatnim o repliki w lokalnym centrum danych. W każdym z tych przypadków przy odczycie jest zwracana wartość z najświeższym znacznikiem czasowym.
3. Poziomy ONE, TWO, THREE zapewniają odpowiednio zapis/odczyt do/z jednej, dwóch i trzech replik. Najbardziej aktualna wersja jest wyznaczana jak wcześniej.

4. Poziom ANY zapewnia, że wartość zostanie zapisana w jakimkolwiek węźle, być może wykorzystując mechanizm przekazania ze wskazaniem (ang. *hinted handoff*). Poziom ten nie ma zastosowania w przypadku operacji odczytu.

Omówione poziomy spójności pozwalają określić kompromis pomiędzy spójnością a dostępnością dla pojedynczych operacji.

4.3. Przykład manipulacji na danych

W dalszej części przedstawiono zapis przykładowej faktury do bazy danych przy użyciu konsoli wiersza poleceń systemu Cassandra. Pierwszym krokiem jest utworzenie przestrzeni kluczy (ang. *keyspace*) o nazwie *FKKeypsace*, która następnie jest ustawiana jako aktywna:

```
create keyspace FKKeyspace;
use FKKeyspace;
```

Ustawienie przestrzeni kluczy jako aktywnej oznacza, że wszystkie kolejne operacje będą jej dotyczyły. W celu umożliwienia zapisu danych konieczne jest utworzenie rodziny kolumn, którą w przykładzie nazwano *Faktury*:

```
create column family Faktury with comparator=UTF8Type and de-
fault_validation_class=UTF8Type and key_validation_class=UTF8Type;
```

Dalej przedstawiono operacje wprowadzające do bazy danych wartości przykładowej faktury nr *123456789*:

```
set Faktury[123456789][nazwaKlienta]='FirmaOdeonSA';
set Faktury[123456789][dataWystawienia]= '2011-11-07';
set Faktury[123456789][opis]='Faktura za ...';
set Faktury[123456789][status]=1;
```

Jako identyfikatora obiektu użyto numeru faktury. Każda z operacji wprowadzania danych (wywołanie polecenia *set*) jest wykonywana atomowo. Podczas wprowadzania danych są definiowane atrybuty obiektu, w konsekwencji zapamiętywane są tylko atrybuty mające wartości. Odczyt wartości wszystkich wprowadzonych atrybutów przykładowej faktury można wykonać poleceniem:

```
get Faktury[123456789];
```

a odczyt nazwy klienta z tej faktury poleceniem:

```
get Faktury[123456789][nazwaKlienta];
```

Podsumowując, system Cassandra oparty jest na modelu klucz-wartość (rozszerzonym o rodziny kolumn), dostarcza prostych atomowych operacji i nie zapewnia wsparcia dla transakcji. Spójność jest określana tylko w kontekście pojedynczej operacji i dotyczy pojedynczej pary klucz-wartość zarówno dla operacji odczytu, jak i zapisu.

5. Oracle NoSQL

Baza danych Oracle NoSQL jest bazą typu klucz-wartość, umożliwiającą przechowywanie danych o rozmiarach rzędu terabajtów przy zachowaniu skalowanej wydajności [16]. Dostępna jest w dwóch edycjach:

a) Community Edition na licencji GNU AFFERO GENERAL PUBLIC LICENSE,
b) Enterprise Edition, jako produkt komercyjny.

W opisywanej wersji 1.2.123 nie ma różnic między tymi edycjami w zakresie funkcjonalności opisywanej w niniejszym artykule.

Dostęp do bazy danych, nazywanej KVStore, udostępnia Oracle NoSQL Database Driver, będący biblioteką Javy. KVStore jest kolekcją węzłów przechowujących (ang. *storage node*), mogących być typowymi komputerami. Każdy z takich węzłów zawiera węzeł replikacji, zawierający jedną lub wiele partycji, przechowujących przypisane im pary klucz-wartość. Węzły replikacji tworzą grupy replikacji, składające się z węzła głównego (ang. *master*) i węzłów replik, przechowujących kopie danych.

5.1. Model danych

Wartości w bazie Oracle NoSQL są przechowywane jako pary klucz-wartość, z którymi jest związany numer wersji. Klucze są listami łańcuchów znaków, a wartości – tabelami bajtów. Serializacja i deserializacja obiektów są pozostawione aplikacjom. W kontekście tematyki niniejszego artykułu istotny jest podział komponentów klucza na wymaganą część główną (ang. *major*) i opcjonalną cześć dodatkową (ang. *minor*). Pary klucz-wartość są przypisywane przez system do partycji i programista nie ma na to wpływu.

5.2. Operacje – atomowość i zachowanie spójności danych

Operacje modyfikujące dane w Oracle NoSQL, tj. operacje *put* i *delete*, są wykonywane w węźle głównym w sposób atomowy i izolowany, co pozwala na uzyskanie własności ACID dla operacji na pojedynczej parze klucz-wartość. W przypadku gdy argumentem operacji *put* jest klucz już istniejący w bazie, związana z nim wartość jest aktualizowana. Operacje modyfikacji wskazanej pary klucz-wartość opcjonalnie umożliwiają pobranie wartości wcześniej związanych z tym kluczem oraz wersji. Możliwe jest również wykonanie grupy operacji w sposób atomowy, jeżeli dotyczą one tylko kluczy o takiej samej części głównej (przykład podano w końcowej części niniejszego rozdziału).

Odczyt danych może zostać wykonany zarówno w węźle głównym, jak i w jednym z węzłów replik, co w przypadku współbieżnie działających programów daje możliwość zwięk-

szenia wydajności kosztem wystąpienia niespójności danych. W celu umożliwienia określenia kompromisu pomiędzy wydajnością a spójnością zostały udostępnione cztery polityki spójności (ang. *consistency policies*), rozumianej jako udostępnianie we wszystkich węzłach takich samych wersji danych:

a) absolutna spójność (ang. *absolute consistency*) – odczyty są wykonywane zawsze w głównym węźle, co zmniejsza wydajność, lecz zapewnia, że zawsze udostępniane są najbardziej aktualne dane,
b) spójność oparta na czasie (ang. *time-based consistency*) – pozwala na określenie maksymalnego tolerowanego opóźnienia synchronizacji repliki, z której następuje odczyt,
c) spójność oparta na numerach wersji (ang. *version-based consistency*) – pozwala określić, że pobierana wartość jest przynajmniej w wersji zadanej w operacji odczytu,
d) brak wymaganej spójności (ang. *none consistency required*) – dane mogą zostać odczytane z repliki niezależnie od stanu jej synchronizacji z węzłem głównym.

Polityka spójności może zostać określona dla magazynu (ang. *store*) lub dla konkretnej operacji. Wyjątek stanowi spójność oparta na numerach wersji, która jest zawsze określana na poziomie operacji.

5.3. Problem współbieżnych aktualizacji

Rozważmy scenariusz współbieżnych aktualizacji, przedstawiony w podrozdziale 3.4. Oracle NoSQL udostępnia wiele operacji, które ułatwiają zapewnienie spójności danych oraz tworzenie w aplikacji namiastek obsługi klasycznych transakcji. W kontekście niniejszego artykułu na szczególną uwagę zasługują operacje *putIfVersion()* i *deleteIfVersion()*, działające na wskazanych wersjach obiektów.

Wykorzystywana w przykładach w niniejszym rozdziale klasa *Faktura* ma postać:

```
public class Faktura implements Serializable {
    private Long nrFaktury;
    private String nazwaKlienta;
    private Date dataWystawienia;
    private String opis;
    private Integer status;
    // … pozostałe dane dziedzinowe
    // … konstruktory oraz metody dostępowe (set i get)
}
```

Najprostszym sposobem zapewnienia spójności złożonego obiektu jest zapisanie całego jego stanu jako wartości związanej z pojedynczym kluczem. W celu ilustracji operacji Oracle NoSQL załóżmy, że dana faktura jest serializowana i przechowywana w postaci jednego obiektu związanego z kluczem wyznaczonym na podstawie jej numeru. Wówczas operacja aktualizacji odczytanej wersji faktury o numerze zadanym zmienną *nrFaktury* wyglądałyby następująco:

```
KVStore store = …; // utworzenie uchwytu do magazynu
// utworzenie głównej części klucza
List<String> majorKeyComponents = new ArrayList<String>();
majorKeyComponents.add(nrFaktury);
Key key = Key.createKey(majorKeyComponents);
// pobranie zserializowanej faktury wraz z wersją
ValueVersion valueVersion = store.get(key);
Faktura faktura = deserializuj(valueVersion.getValue());
Version wersjaFaktury = valueVersion.getVersion();
// modyfikuj fakturę
// zapisz zmodyfikowaną fakturę, jeżeli w między czasie nie uległa zmodyfikacji
Version nowaWersjaFaktury = store.putIfVersion(key, serializuj(faktura),
                                                        wersjaFaktury);
```

W celu uproszczenia przykładu użyto hipotetycznych metod *serializuj* i *deserialiuj* przekształcających odpowiednio fakturę na tablicę bajtów i tablicę bajtów na fakturę. Metoda *putIfVersion()* zwraca nową wersję obiektu lub *null*, jeżeli obiekt w żądanej wersji nie istniał.

Dalej podano przykład odczytu faktury w zadanej wersji:

```
Consistency consistency = new Consistency.Version(wersjaFaktury, 10,
                                                  TimeUnit.MILLISECONDS);
ValueVersion valueVersion = store.get(key, consistency, 1, TimeUnit.SECONDS);
Faktura faktura = deserializuj(valueVersion.getValue());
```

Przy implementacji transakcji w aplikacjach pomocne mogą się okazać atomowe operacje warunkowe *putIfAbsent()* oraz *putIfPresent()*, z których pierwsza dodaje nową parę klucz-wartość, a druga aktualizuje istniejącą.

Oracle NoSQL udostępnia również operacje na grupach kluczy. Z punktu widzenia celów niniejszego artykułu ważne jest rozróżnienie operacji atomowych *multiGet* i *multiDelete* od nieatomowych *multiGetIterator()* i *storeIterator()*. W przypadku pierwszej grupy wartości muszą współdzielić główną część klucza i są pobierane od razu do pamięci operacyjnej. W przypadku drugiej grupy wartości są pobierane w miarę potrzeb do pamięci w wersjach, w jakich są aktualnie przechowywane w bazie danych.

5.4. Gwarancje trwałości

Wszystkie operacje modyfikujące bazę mają wersje określające gwarancje trwałości (ang. *durability guarantees*), tzn. politykę określającą jak pewny jest zapis w przypadku wystąpienia poważnej awarii. Składają się na nią gwarancje zapisu danych (możliwe opcje: zapis na dysku, zapis w buforach systemu plików lub zapis w pamięci operacyjnej) oraz gwarancje potwierdzenia zapisu w replikach (możliwe opcje: zapis we wszystkich replikach, zapis w większości replik, nie jest wymagane potwierdzenie z replik).

W dalszej części pokazano modyfikację poprzedniego przykładu, w której przy zapisie wymaga się, aby w węźle głównym dane zostały zapisane na dysk, w węzłach replik nie jest

to wymagane, a decyzja o replikacji nowej wartości jest podejmowana na podstawie zwykłej większości. Przy zapisie są pobierane poprzednia wartość faktury i jej wersja [15]. Ograniczenie czasowe na wykonanie operacji zostało zadane na 2 sekundy.

```
Durability durability = new Durability(Durability.SyncPolicy.SYNC,
    Durability.SyncPolicy.NO_SYNC, Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
ReturnValueVersion poprzedniaWartoscFaktury = new ReturnValueVersion(
                                                ReturnValueVersion.Choice.VALUE);
store.putIfVersion(key, value, wersjaFaktury, poprzedniaWartoscFaktury,
                                                durability, 2, TimeUnit.SECONDS);
```

5.5. Atomowe wykonanie grupy operacji

Sekwencje operacji pozwalają na atomowe wykonywanie grupy operacji modyfikujących, które odnoszą się do obiektów o takiej samej głównej części klucza. Sekwencje operacji są wykonywane w izolacji, co oznacza, że inne, współbieżnie obsługiwane żądania nie będą widzieć wykonywanych zmian ani nie będą mogły modyfikować tych samych danych, dopóki sekwencja się nie zakończy. Ponieważ sekwencje operacji obejmują tylko operacje modyfikujące dane, nie jest możliwe odczytanie spójnego stanu bazy danych na określony moment w czasie.

Dalej zamieszczono przykład zapisu faktury jako atomowo wykonywanej sekwencji operacji. Przyjęto, że klucze mają postać:

```
<rok>/<miesiąc>/<nrFaktury>—<nazwa_atrybutu>
```

gdzie znak „–” oddziela część główną klucza od jego części pomocniczej. Dla przykładowej faktury daje to klucze:

```
/2011/11/123456789/—/nazwaKlienta
/2011/11/123456789/—/dataWystawienia
/2011/11/123456789/—/opis
/2011/11/123456789/—/status
```

W zamieszczonym dalej przykładowym kodzie pominięto powtarzające się operacje dla kolejnych atrybutów. Miesiąc i rok są wyznaczane na podstawie daty wystawienia faktury.

```
Faktura faktura = …; // jak we wcześniejszych przykładach
KVStore store = …; // utworzenie uchwytu do magazynu
// utworzenie głównej części klucza wspólnej dla wszystkich atrybutów
List<String> majorKeyComponents = new ArrayList<String>();
majorKeyComponents.add(faktura.getRok().toString());
majorKeyComponents.add(faktura.getMiesiac().toString());
majorKeyComponents.add(faktura.getNrFaktury().toString());
// utworzenie pomocniczych części klucza dla atrybutu nazwaKlienta
List<String> minorKeyComponentsNazwaKlienta = new ArrayList<String>();
minorKeyComponentsNazwaKlienta.add("nazwaKlienta");
Key keyNazwaKlienta = Key.createKey(majorKeyComponents,
                                                minorKeyComponentsNazwaKlienta);
// utworzenie wartości z nazwą klienta
Value valueNazwaKlienta =
                        Value.createValue(faktura.getNazwaKlienta().getBytes());
```

```
// utworzenie kluczy i wartości dla atrybutów dataWystawienia, opis i status
…
// Utworzenie sekwencji operacji
OperationFactory operationFactory = store.getOperationFactory();
List<Operation> operations = new ArrayList<Operation>();
operations.add(operationFactory.createPut(keyNazwaKlienta, valueNazwaKlienta));
operations.add(operationFactory.createPut(keyDataWystawienia,
                                                    valueDataWystawienia));
operations.add(operationFactory.createPut(keyOpis, valueOpis));
operations.add(operationFactory.createPut(keyStatus, valueStatus));
// Wykonanie sekwencji operacji – dla każdej operacji jest zwracany jej rezultat
List<OperationResult> results = store.execute(operations);
```

Podsumowując, system Oracle NoSQL oparty jest na modelu klucz-wartość, w którym klucz może mieć budowę segmentową. Modyfikacje pojedynczej pary klucz-wartość oraz grupy par klucz-wartość współdzielących główną część klucza mogą być wykonywane przy zachowaniu własności ACID. Poziomy spójności dotyczą tylko operacji odczytu. Dostępne operacje umożliwiają implementację zarządzania transakcjami w aplikacji w taki samsposób analogiczny jak w rozdziale poświęconym MongoDB (implementacja protokołu dwufazowego).

6. Podsumowanie

W przedstawionym artykule przedmiotem analizy są systemy zarządzania bazami NoSQL z modelem typu klucz-wartość oraz bazą dokumentową. Konkretne systemy, jak: MongoDB, Windows Azure, Cassandra, Oracle NoSQL – zostały wybrane w sposób subiektywny, na podstawie ich popularności (m.in. liczby wzmiankowań w Internecie). Jednak zadbano o to, by uwzględnić zarówno produkty typu open-source (Cassandra, MongoDB, Oracle NoSQL Community Edition), jak i komercyjne (Windows Azure, Oracle NoSQL Enterprise Edition).

Wspólnym mianownikiem omawianych podejść jest nazwa bazowej koncepcji, tj. modelu danych klucz-wartość. Jednak poza tym hasłem podejścia te różnią się przez wprowadzenie własnych, unikalnych rozwiązań, np.: dwuczłonowego typu klucza (Azure), możliwość zmiany wartości klucza w istniejącej encji (Azure), możliwości grupowego zatwierdzania zmian w ograniczonym zakresie (Azure, MongoDB, Oracle NoSQL), wartość klucza może być tablicą/listą (MongoDB, Oracle NoSQL), wartością klucza może być dokument zagnieżdżony (MongoDB) i innych.

Ważne wydaje się spostrzeżenie, że omówione podejścia charakteryzują się dość różnym API, umożliwiającym dostęp do bazy NoSQL. Z pewnością nie można tu mówić o jakimś ogólnym standardzie, co utrudnia programowanie z użyciem różnych baz NoSQL, nie wspominając o przenośności rozwiązań.

Mechanizmy zapewnienia spójności w ramach współbieżnego dostępu czy zapewnienia transakcyjnego przetwarzania są słabo rozwinięte w systemach NoSQL. Oczywiście jest to

zamierzone i zgodne z podejściem, w którym szybkość przetwarzania jest elementem najistotniejszym, a obsługa spójności danych może być w dużej części implementowana w ramach kodu samej aplikacji wykorzystującej dane (implementacja zatwierdzania dwufazowego, kompensacje transakcji itp.). Atomowość przetwarzania tylko pojedynczych obiektów klucz-wartość na ogół jest zachowana we wszystkich systemach (ale np. w MongoDB atomowość nie jest dostępna w systemie rozproszonym, a jedynie dla pojedynczej instancji serwera). W kilku systemach istnieją pewne rozszerzenia (częściowo wspomagające programistę, tzn. wyręczające go z konieczności złożonej obsługi sytuacji awaryjnych), np. grupowe transakcje w Azure, Oracle NoSQL czy izolacja grupowych operacji w nierozproszonym systemie MongoDB. Nie ma pełnej implementacji transakcyjnego przetwarzania w sensie paradygmatu ACID.

BIBLIOGRAFIA

1. http://downloads.mongodb.org/docs/mongodb-docs-2012-01-16.pdf.
2. Licensing – MongoDB (2012), http://www.mongodb.org/display/DOCS/Licensing.
3. GNU Affero General Public License – GNU (2012, http://www.gnu.org/licenses/agpl-3.0.html.
4. Perform Two Phase Commits. The MongoDB Cookbook (2012), http://cookbook.mongodb.org/patterns/perform-two-phase-commits.
5. ACID (2012), http://en.wikipedia.org/wiki/ACID.
6. Tour – Overview – Windows Azure (2012), http://www.windowsazure.com/en-us/home/tour/overview.
7. Cloud Storage – Windows Azure (2012), http://www.windowsazure.com/en-us/develop/net/fundamentals/cloud-storage.
8. Haridas J., Nilakantan N., Calder B.: Windows Azure Table, http://www.scribd.com/doc/63485303/Windows-Azure-Table-May-2009.
9. Biesiada D., Cichocki P., Kopacz T., Zass B., Żarski A., Żyliński M.: Windows Azure – Platforma Cloud Computing dla programistów. APN Promise, Warszawa 2010.
10. Performing Entity Group Transactions (2012), http://msdn.microsoft.com/en-us/library/dd894038.aspx.
11. RESTful web services (2012), http://en.wikipedia.org/wiki/Representational_state_transfer.
12. Table Service REST API (2012), http://msdn.microsoft.com/en-us/library/windows-azure/dd179423.aspx.

13. Set transaction isolation level (2012), http://msdn.microsoft.com/en-us/library/ms173763.aspx.
14. Tiwari S.: Professional NoSQL. Wrox, 2011.
15. Getting Started with NoSQL Database 11g Release 2, Library Version 11.2.1.1, Oracle 2011.
16. Oracle NoSQL Database Administrator's Guide 11g Release 2, Library Version 11.2.1.1, Oracle 2011.
17. DataStax Cassandra 1.0 Documentation (2012), http://www.datastax.com/docs/1.0/ddl.
18. Apache Cassandra Glossary (2012), http://io.typepad.com/glossary.html.
19. API – Cassandra Wiki (2012), http://wiki.apache.org/cassandra/API.

Wpłynęło do Redakcji 31 stycznia 2012 r.

Abstract

New database solutions collectively named NoSQL are a new approach to data storage and processing dedicated for cloud computing. These approaches are internally simplified solutions comparing to the ones based on RDBMS-es.

The problem of consistency preservation is globally important in data processing, so it's especially important for NoSQL-based systems which may operate in distributed environments. NoSQL databases efficiency is a major feature. It was partially achieved at the expense of some difficulties of programming. NoSQL-based applications should handle some situations which may introduce inconsistency in data.

The paper presents selected open source and commercial NoSQL systems mainly based on the broadly-understood key-value concept. Particular methods of data manipulation were described using source code examples.

The simple methods of data consistency preservation for MongoDB, Mircosoft Windows Azure, Apache Cassandra, Oracle NoSQL were considered. The simple source codes of programs for consistency handling were shown too.

Adresy

Dariusz R. AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, draugustyn@polsl.pl.

Piotr BAJERSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, piotr.bajerski@polsl.pl.

Robert BRZESKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, robert.brzeski@polsl.pl.