Indeksy w JLoXiM-ie

Michał Lenart

8.IV.2010

Plan prezentacji

Wstęp

Przegląd API
- Store
- Indeksy
- Zarządzanie indeksami

Implementacja

Bibliografia

Założenia

Co to jest indeks

Struktura danych tworzona w celu przyspieszenia wykonania zapytań.

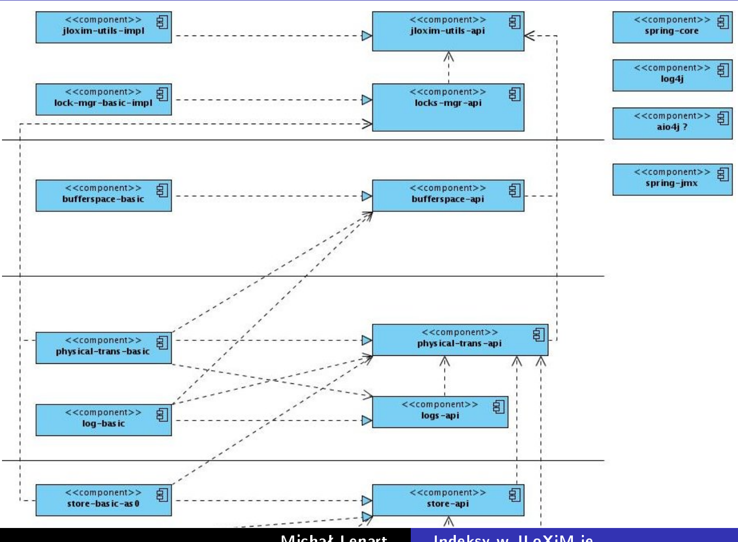
- opisywane tutaj, to indeksy pomocnicze w JLoXiM: mapowanie klucz -> OID obiektu
- konieczny dodatkowy narzut przy modyfikacjach danych
- zajmuje dodatkowe miejsce na dysku

Zastosowania

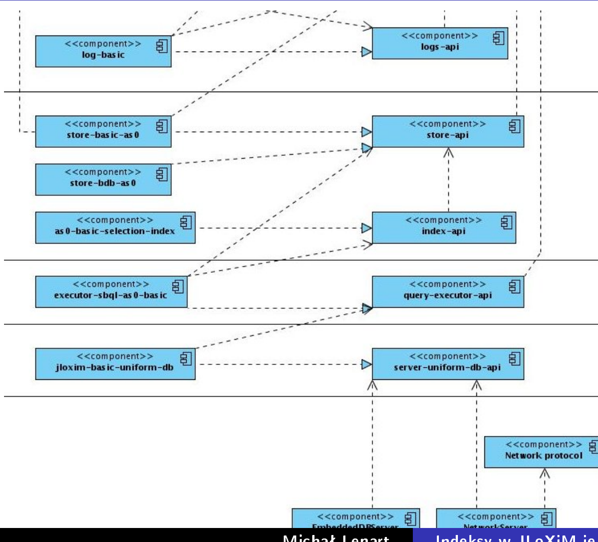
Przykłady zapytań, w których możliwe, że warto zastosować indeksy:

- najprostszy przykład:
  Osoba ***where*** *(nazwisko = "Kowalski")*
- pytania zakresowe:
  Osoba ***where*** *(wiek > 18* ***and*** *wiek < 70)*

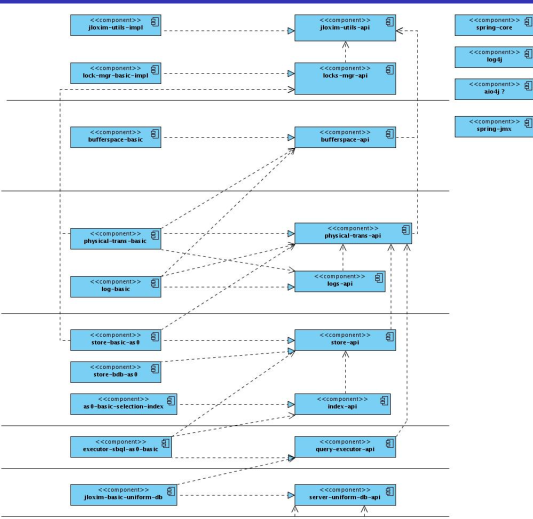
Miejsce w architekturze projektu JLoXiM



Miejsce w architekturze projektu JLoXiM



Miejsce w architekturze projektu JLoXiM



Co indeksujemy - przykład

- wartości w indeksie to obiekty o ścieżce *people.person*
- wyszukiwanie po imieniu - klucze to obiekty atomowe o ścieżce *name* (względem wartości)
- wyszukiwanie po adresie - klucze to pary (ogólniej - krotki) o ścieżkach (odpowiednio) *address.city* oraz *address.street*

```
<people>
  <person>
    <name type="string">Kowalski</name>
    <address>
      <city type="string">Warsaw</city>
      <street type="string">Banacha</zip>
    </address>
  </person>
  <person>
    <name type="string">Nowak</name>
    <address>
      <city type="string">Warsaw</city>
      <street type="string">Pasteura</zip>
    </address>
  </person>
</people>
```

Wymagania

Założenia, jakie powinna spełniać implementacja indeksów dla JLoXiM:

- wygodne i funkcjonalne API
- automatycznie zachowanie spójności danych - proponowane rozwiązanie - operacje na indeksie wykonywane w ramach tej samej transakcji co operacje na Store
- jak najmniejszy narzut na modyfikacje danych
- rozszerzalność - możliwość korzystania z różnych implementacji struktur używanych do indeksowania (B-drzewa, B+drzewa, tablice z haszowaniem) jak również różnych Store-ów
- ma działać w modelu AS0 z opcjonalnym uwzględnianiem schematu.

Wymagania dotyczące Store-a

- żeby indeks sam "się uaktualniał", musi "wiedzieć" o operacjach CRUD wykonywanych na Store
- Store musi więc powiadamiać indeks o wszystkich aktualizacjach
- najbardziej oczywiste rozwiązanie - wzorzec Obserwator
- interfejsy realizujące ten wzorzec: PathNamesObservingStoreAS0 i PathNamesModificationListener (mogą też być przydatne w implementacji schematów w JLoXiM)

PathNamesModificationListener

Metody zdefiniowane w interfejsie PathNamesModificationListener:

```
void beforeAtomicValSet(
  Transaction t, List<Integer> path,
  AbstractOid oid, AtomicValue newVal)

void afterAtomicValSet(
  Transaction t, List<Integer> path,
  AbstractOid oid, AtomicValue newVal)

void beforeAddSubobject(
  Transaction t, List<Integer> path,
  AbstractOid oid, AS0ObjectEditable object)

void afterAddSubobject(
  Transaction t, List<Integer> path,
  AbstractOid oid, AS0ObjectEditable object)
```

PathNamesModificationListener

```
void beforePointerDestSet(...)

void afterPointerDestSet(...)

void beforeSetComplexObjectValue(...)

void afterSetComplexObjectValue(...)

void beforeMoveObject(
  Transaction t, List<Integer> path, List<Integer> newPath,
  AbstractOid oid)

void afterMoveObject(
  Transaction t, List<Integer> path, List<Integer> newPath,
  AbstractOid oid)

void beforeRemoveObject(
  Transaction t, List<Integer> path, AbstractOid oid)
```

PathNamesObservingStoreAS0

Metody zdefiniowane w interfejsie PathNamesObservingStoreAS0:

```
void setBackingStore(StoreAS0 store)

void registerListener(
  List<Integer> path,
  PathNamesModificationListener listener)

void unregisterListener(PathNamesModificationListener listener)

void unregisterListeners(List<Integer> path)

void unregisterAllListeners()
```

PathNamesObservingStoreAS0 - problem ze ścieżkami

Weźmy pod uwagę następującą sytuację (SZBD musi ją uwzględnić):

- obiekty *F*, *X* i *Y* są odpowiednio na ścieżkach: *A.B.C.D.E.F*, *X* i *Y*
- w transakcji T1 przenosimy *B* z *A* do *X*
- w transakcji T2 przenosimy *D* z *C* do *Y*
- w transakcji T3 zmieniamy wartość *F* z *"V1"* na *"V2"*
- T3 musi powiadomić odpowiedniego listenera o zmianie...
- ... **problem** - na jakiej ścieżce jest *F*??
- wędrując do korzenia możemy się dowiedzieć, że *F* jest na *A.B.C.D.E.F*, *X.B.C.D.E.F* lub *Y.D.E.F*

PathNamesObservingStoreAS0 - problem ze ścieżkami

Rozwiązanie: PathNamesObservingStoreAS0 musi mieć możliwość blokowania obiektów w ramach transakcji.

- problem - jeśli będzie próbował to robić niezależnie od Store-a możemy stracić możliwość wykrywania zakleszczeń
- rozwiązanie - rozszerzenie interfejsu StoreAS0 o metodę:

```
void getLogicalLock(
    Transaction t, AbstractOid OID, LockTypeEnum type)
```

- przy wykonywaniu moveObject i removeObject bierzemy blokadę PROTECTED_WRITE dla obiektu i jego poddrzewa.
- przy pozostałych operacjach, przy których wywołujemy listenerów - blokada PROTECTED_READ na obiekcie.

KeyAndOidPair

KeyAndOidPair - propozycja typu wyniku wyszukiwania na indeksie:

```
class KeyAndOidPair {
    AbstractOID oid;
    AS0Resultset key;
}
```

Index

Metody zdefiniowane w interfejsie Index:

```
ClosableIterator<AbstractOid> search(
    Transaction t,
    AS0ResultSet key);

ClosableIterator<KeyAndOidPair> searchWithPrefix(
   Transaction t,
   AS0Resultset exactFields,
   AS0Resultset prefixFields);

ClosableIterator<KeyAndOidPair> searchRange(
    Transaction t,
    AS0ResultSet fromKey, boolean leftInclusive
    AS0ResultSet toKey, boolean rightInclusive);
```

Index

Ponadto interfejs Index definiuje metody do administrowania indeksem oraz sprawdzania jego właściwości (co indeksuje i jakie operacje udostępnia):

```
String getIndexId();

List<List<Integer>> getKeyPaths();
List<Integer> getObjectsPath();

boolean isRangeSearchSupported();

boolean isActive();
void rebuild();
```

Index - przykład

Dla indeksu idx, w którym:

- objectsPath = *firma.osoba*
- keyPaths = [*nazwisko*, *adres.miasto*, *adres.ulica*]

Szukamy:

- wszystkich Kowalskich mieszkających w Warszawie przy ul. Banacha:

```
idx.search(t, ["Kowalski", "Warszawa", "Banacha"])
```

- wszystkich Kowalskich mieszkających gdziekolwiek:

```
idx.search(t, [nazwisko="Kowalski"])
idx.search(t, ["Kowalski"])
```

Index - czego się nie da

Nie można wykonywać zapytań postaci: idx.search(t, [*, klucz])

- Każde z takich zapytań skończy się wyjątkiem:

```
idx.search(t, [nazwisko->"Kowalski",
               adres.ulica->"Banacha"])

idx.search(t, [adres.miasto->"Warszawa"])
```

- wykonywanie takich zapytań przy leksykograficznym porządku na kluczach złożonych nie jest wydajne

Index - jak to jest widziane na zewnątrz

- aktualizacje indeksów będą niewidoczne dla użytkownika
- nie przewiduję implementowania optymalizatora zapytań automatycznie wykorzystującego indeksy
- za to musi być możliwe dodanie go w przyszłości

Index - jak to jest widziane na zewnątrz

Propozycja udostępnienia indeksu bez optymalizatora:

- tworzymy indeks:
  ```
  CREATE INDEX idx1 ON firma.osoba
  VALUE (nazwisko, adres.miasto, adres.ulica)
  ```
- wtedy w systemie (na stosie) pojawiają się funkcje:
  ```
  INDEXES.idx1.search(nazwisko, miasto, ulica)
  ```
- chcąc wyszukać wszystkich Kowalskich z Warszawy, wywołujemy:
  ```
  INDEXES.idx1.search("Kowalski", "Warszawa")
  ```

IndexingStructure

- w uproszczeniu: mapa klucz->(OID, klucz), czyli jest to "silnik" indeksu
- implementacja może być B-drzewem, tablicą z haszowaniem itp.
- executor widzi interfejs Index, który udostępnia wyszukiwanie ale nie umożliwia modyfikacji
- przy operacjach CRUD PathNamesModificationListener aktualizuje zawartość indeksu poprzez IndexingStructure

IndexingStructure

Metody zdefiniowane w interfejsie IndexingStructure:

```
ClosableIterator<AbstractOid> get(
    Transaction t,
    List<AtomicValue> key);

ClosableIterator<KeyAndOidPair> getRange(
    Transaction t,
    List<AtomicValue> fromKey, boolean leftInclusive
    List<AtomicValue> toKey, boolean rightInclusive);

void put(
    Transaction t,
    List<AtomicValue> key, AbstractOid oid);
```

IndexingStructure

Metody zdefiniowane w interfejsie IndexingStructure cd:

```
void changeKeyValue(Transaction t, KeyAndOidPair keyOid,
    List<AtomicValue> newKey)

void remove(Transaction t, KeyAndOidPair keyOid);

boolean isRangeSearchSupported();

void clear();

void close();

void destroy();
```

IndexManager

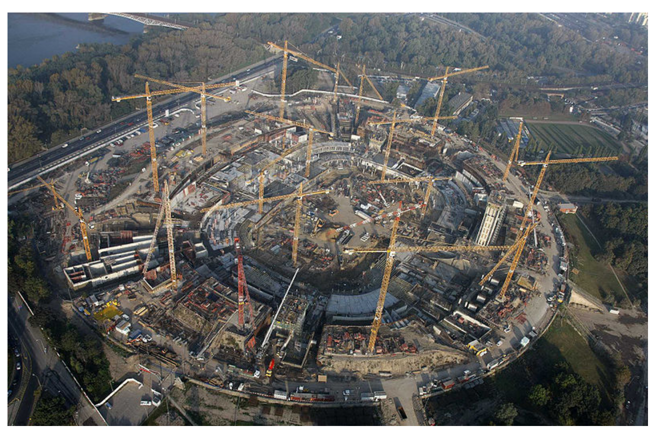
- Zarządca indeksów - metody:

```
List<Index> getIndexes(
    Transaction t,
    List<Integer> objPath);

public void createIndex(
    List<Integer> objPath,
    List<List<Integer>> keyPaths,
    IndexType type);

void rebuildAllIndexes();
void removeIndex(String indexId);
...
```

Indeksy - uwagi

- wyszukiwanie w indeksach odbywa się w tej samej transakcji, co operacje na Store
- to zapewnia integrację danych
- do administrowania indeksami nie trzeba tworzyć transakcji
- i tak muszą być tworzone nowe, "wewnętrzne" transakcje (przebudowywanie indeksu może się nie udać w ramach jednej transakcji)

Implementacja



Struktury użyte do indeksowania

- w tej chwili powstaje implementacja oparta na Berkeley DB (czyli B-drzewie)
- zalety - stabilność, dojrzałość tego projektu, coś co na pewno już działa
- w planach dodanie implementacji opartej o natywne B-drzewo - projekt trans-btree-impl tworzony przez Kamila Adamczyka

Jak dodać nowy typ indeksu?

- napisać własną implementację klasy IndexingStructure
- dodać wpis do IndexType (np. *MOJE_NOWE_B_DRZEWO*)
- zaimplementować odpowiednią fabrykę (IndexingStructureFactory) i podpiąć ją do IndexManager-a w konfiguracji w Springu.

Modyfikacje na indeksie

Przykładowy indeks:

```
CREATE INDEX idx1 ON firma.osoba
VALUE (adres.miasto, adres.ulica)
```

Rejestrujemy 2 typy listenerów

- ObjectModificationListener - na ścieżkach *firma* oraz *firma.osoba*. Reaguje na dodawanie/usuwanie obiektów, na których jest indeks
- KeyModificationListener - na ścieżkach *firma.osoba.adres*, *firma.osoba.adres.miasto* oraz *firma.osoba.adres.ulica* - reaguje na modyfikacje kluczy indeksu

IndexManagerImpl

Implementacja zarządcy indeksów (IndexManager-a) - problem - jak jakaś transakcja używa (lub chociaż wie o istnieniu) indeksu, nie może on zostać w tym czasie zdeaktywowany lub usunięty.

Najprostsze rozwiązanie?

- skorzystać z dodatkowego LockManager-a.
- każda *aktywna transakcja*, która wykonała metodę getIndexes(...) trzyma blokadę CONCURRENT_READ
- transakcja, która chce znaleźć indeks musi dostać (na czas szukania) PROTECTED_READ
- każdy *wątek* próbujący usunąć indeks musi dostać blokadę EXCLUSIVE
- każdy *wątek* dodający indeks musi dostać blokadę PROTECTED_WRITE

IndexManagerImpl

Wady przedstawionego rozwiązania?

- aby usunąć indeks trzeba poczekać na zakończenie wszystkich aktywnych transakcji (korzystających z indeksów) oraz zablokować wykonywanie nowych (bo blokada EXCLUSIVE)
- **rozwiązanie?** - można pamiętać mapę (transakcja->widziane przez nią indeksy) i blokować poszczególne indeksy a nie całego IndexManager-a
- wówczas trochę większy narzut przy normalnym działaniu
- Ale tak naprawdę - jak często usuwamy indeksy? Czy jest się czym przejmować?

Bibliografia

- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom.
  Systemy baz danych
  Pełny wykład.
  WNT 2006
- Tomasz Marek Kowalski.
  Transparent Indexing in Distributed Object-Oriented Databases.
  Łódź 2009
- PostgreSQL Documentation
  http://www.postgresql.org/docs/