Masarykova univerzita
Fakulta informatiky

Relační přístup k Amazon SimpleDB

Diplomová práce

Radim Hopp

Brno, Podzim 2013

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Radim Hopp

Vedoucí práce: Mgr. Marek Grác

Poděkování

Rád bych poděkoval vedoucímu práce Mgr. Marku Grácovi za ochotný přístup a věnovaný čas, panu Lukáši Tinklovi a lidem z komunity KDE na IRC kanále #kde-devel za cenné rady.

Shrnutí

Cílem této práce bylo důkladné prozkoumání stavu Kiosk Frameworku v KDE 4.x, zjištění stavu nástroje kiosktool v KDE 4.x a doplnění jeho funkčnosti.

Klíčová slova

KDE, Kiosk Framework, kiosktool, Qt

Obsah

1 **Úvod** . . . 3
2 **JBoss Teiid** . . . 4
2.1 *Části JBoss Teiid* . . . 4
3 **Amazon SimpleDB** . . . 6
3.1 *Struktura databáze SimpleDB* . . . 6
3.2 *Vlastnosti databáze Amazon SimpleDB* . . . 7
3.2.1 SimpleDB jako služba . . . 7
3.2.2 Flexibilita . . . 7
3.2.3 Indexování . . . 7
3.2.4 Konzistence . . . 7
Příklad konzistence . . . 8
3.2.5 Omezení databáze SimpleDB . . . 9
4 **Komunikace s databází Amazon SimpleDB** . . . 10
4.0.6 Přehled příkazů SimpleDB . . . 10
4.0.7 Podmíněné vkládání a mazání dat . . . 10
4.0.8 Bezpečnost . . . 11
Proces autentizace . . . 11
4.0.9 Knihovny pro komunikaci . . . 11
5 **Mapování Teiid dotazů na SimpleDB dotazy** . . . 12
5.1 *Vícehodnotové atributy* . . . 12
5.2 *SELECT* . . . 12
5.3 *INSERT* . . . 13
5.4 *UPDATE* . . . 13
5.5 *DELETE* . . . 14
6 **Implementace** . . . 15
6.1 *Konektor* . . . 15
6.1.1 Modul simpledb-api . . . 15
6.1.2 Modul connector-simpledb . . . 16
6.2 *Překladač* . . . 17
6.2.1 SimpleDBExecutionFactory . . . 17
6.2.2 Executory . . . 20
6.2.3 Visitory . . . 21
Návrhový vzor Visitor . . . 21
Jazykový visitor v Teiidu . . . 22
SimpleDBSQLVisitor . . . 23
SimpleDBInsertVisitor . . . 23
SimpleDBUpdateVisitor . . . 24

7 **Tutoriál** 26
8 **Závěr** 27

1 Úvod

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec blandit turpis sit amet nibh volutpat ultrices. Quisque ut placerat quam. Nunc pharetra non metus in vestibulum. In in leo porta, rutrum nisi a, scelerisque dolor. Donec non vehicula dui. Ut ut urna quis massa sollicitudin laoreet sed et sapien. Quisque pellentesque massa quam, vitae vulputate enim venenatis id. Curabitur ornare enim leo, et venenatis orci interdum id. Vivamus nunc augue, dictum eu semper consequat, vestibulum vitae eros. Phasellus sodales tincidunt odio, vel rhoncus orci dignissim eu.

Curabitur urna purus, varius id mattis adipiscing, aliquam et sem. Duis sed enim non nisl commodo molestie. Vestibulum gravida suscipit lectus vitae interdum. Maecenas quis vestibulum nibh. Nunc vel porta massa. Sed interdum tortor ac elit sollicitudin, ut auctor nisi ullamcorper. Integer nec eros tempus arcu convallis molestie consectetur ac nunc.

Pellentesque non posuere quam, et ultricies odio. Phasellus varius arcu ac arcu elementum venenatis. Aliquam tempor elit et urna suscipit facilisis. Quisque id ligula nec justo malesuada egestas vel eget nulla. Phasellus non bibendum sem. Ut venenatis quam id nibh cursus, ac convallis metus adipiscing. Integer imperdiet, lorem et ullamcorper pellentesque, ligula mi auctor elit, ut suscipit purus lorem in nibh. Ut aliquam nibh non adipiscing dignissim. Duis consectetur dui ac turpis semper, non lacinia metus feugiat. Aliquam malesuada congue felis eu luctus.

Nam id volutpat metus, vitae sodales enim. Nullam tempor odio libero, id iaculis magna pellentesque accumsan. Vestibulum accumsan mauris quis urna sodales, placerat imperdiet metus tincidunt. Nullam consequat purus sit amet purus consectetur porttitor. Praesent fringilla vitae turpis rutrum rhoncus. Aenean tincidunt, mauris in mattis ornare, dui elit lobortis enim, et elementum libero nulla ut sapien. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse commodo leo placerat rhoncus adipiscing. Etiam rhoncus rhoncus interdum. Sed sed nibh ut sapien varius sodales. Maecenas lacinia nunc eu neque tempus aliquet.

Vestibulum tempus laoreet lectus. Mauris accumsan velit ac erat condimentum pellentesque. Suspendisse et lorem vitae lorem volutpat interdum. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Sed ac tincidunt leo, sed vulputate mi. Maecenas et nulla egestas, adipiscing neque a, blandit massa. Nam ullamcorper, tortor id accumsan consectetur, tortor neque ultrices mi, a laoreet odio ipsum et tellus. Duis lobortis faucibus luctus.

2 JBoss Teiid

JBoss Teiid je otevřený software pro virtualizaci dat z více zdrojů. Tento projekt je zaštiťován v rámci JBoss komunitních projektů[1] spolu s projekty jako WildFly nebo GateIn[2].

Firma Red Hat nabízí také plně podporovanou a certifikovanou verzi pod názvem Red Hat JBoss Enterprise Data Services Platform. Tento produkt kromě projektu Teiid v sobě obsahuje také JBoss Enterprise Application Platform a další produkty[3]

2.1 Části JBoss Teiid

- **Query engine**

  Jedná se o srdce projektu, které zpracovává relační, XML, XQuery a jiné dotazy ze zdrojů dat. Také zajišťuje podporu pro schémata vytvořené z jednoho, či z více různých zdrojů dat. V neposlední řadě se query engine stará o transakce a uživatelem definované funkce.

- **JDBC ovladač**

  JDBC ovladač slouží pro jednoduché použití Teiidu v ostatních java aplikacích.

- **Server**

  Tato část je zodpovědná za běh query enginu v rámci aplikačního serveru (WildFly nebo Red Hat JBoss Enterprise Application Platform) a zaručuje škálovatelnost a snadnou správu. Součástí je administrační konzole, ve které je možno jednoduše nastavit například deploynutou virtuální databázi nebo nastavení vláken query enginu.

- **Konektory a překladače**

  Teiid přichází s řadou konektorů a překladačů, díky kterým je schopen napojit se na řadu různých zdrojů dat. Konektory zajišťují napojení na zdroj dat, překladače potom obstarávají získávání dat podle Teiid

1. http://www.jboss.org/overview/
2. kompletní seznam projektů na http://www.jboss.org/projects
3. kompletní seznam produktů a projektů obsažených v Red Hat JBoss Enterprise Data Services Platform i s použitými verzemi k naleznutí na https://access.redhat.com/site/articles/112333

požadavků. Základním kamenem jsou konektory pro většinu relačních databází, webové služby, textových souborů a LDAP[4].

Cíl této diplomové práce je rozšířit tuto sekci o connector pro NoSQL databázi Amazon SimpleDB.

- **Nástroje**
  - Teiid Designer

    Teiid Designer je separátní projekt (není součástí projektu JBoss Teiid), který ovšem také patří mezi komunitní projekty jboss.org. Jedná se o sadu zásuvných modulů pro vývojové prostředí Eclipse, které usnadní definování virtuálních databází (pohledy, procedury, ...)
  - Nástroje pro monitorování a správu

    Dvěmi hlavními nástroji pro monitorování a správu jsou Teiid Web Console (V zásadě jde o rozšíření webové konzole aplikačního serveru o položky pro správu a monitorování Teiid instance) a zásuvný modul pro RHQ[5] pro kontrolu více serverů či clusterů Teiidu.
  - Skriptování

    V rámci Teiidu je také distribuován Teiid AdminShell, což je skriptovací nástroj založený na jazyku Groovy. AdminShell umožňuje spravovat Teiid z příkazové řádky či pomocí skriptu, což uživateli velmi usnadní provádění často se opakujících činností.

4. Lightweight Directory Access Protocol)
5. RQH je komunitní projekt vedený pod jboss.org pro správu více serverů, http://www.jboss.org/rhq

3 Amazon SimpleDB

SimpleDB je databáze z dílen firmy Amazon a patří do skupiny produktů Amazon Web Services. Tato databáze je takzvaná NoSQL, což znamená, že data v ní jsou uložená ve formátu, jaký není běžný pro relační databáze (tabulky, sloupce, řádky). SimpleDB je poskytována firmou Amazon jako služba. To znamená, že není možné nasadit si databázi SimpleDB na vlastním serveru, ale běží pouze na serverech firmy Amazon.

3.1 Struktura databáze SimpleDB

Struktura SimpleDB databází je možno přirovnat k tabulkovému procesoru.

- **Uživatelský účet (Customer Account)**

  V rámci jednoho účtu lze mít více domén stejně jako v rámci jednoho souboru tabulkového procesoru je možno mít více listů s tabulkami.

- **Doména (Domain)**

  Analogie k jedné tabulce v tabulkovém procesoru. Obsahuje řádky (položky), sloupce (atributy) a jednotlivé buňky (hodnoty).

- **Položky (Items)**

  Řádky tabulky. Reprezentují jednotlivé objekty uložené v databázi s jedním, či více vyplněnými atributy.

- **Atributy (Attributes)**

  Sloupce tabulky. Určují kategorie dat, jakých můžou položky nabývat.

- **Hodnoty (Values)**

  Hodnoty si lze představit jako jednotlivé buňky tabulky. Tady se ovšem analogie s tabulkovým procesorem rozchází, neboť jedna „buňka“ může nabývat více hodnot najednou.

Navíc má každý prvek domény (řádek tabulky) povinný atribut `ItemName`, který musí být v dané doméně unikátní.

SimpleDB nepoužívá žádné jiné datové typy krome typu `String`, takže i čísla jsou v databázi uloženy a je s nimi nakládáno jako s textovými řetězci. To s sebou nese řadu problémů a hlavně zátěž na vývojáře aplikace tyto problémy řešit. Typickým příkladem takového problému je, že čísla nejdou řadit lexikograficky (lexikograficky: 10<2, což je špatně). Řešením je doplnění nul před čísla (00010>00002, což je správně).

3.2 Vlastnosti databáze Amazon SimpleDB

3.2.1 SimpleDB jako služba

Databáze Amazon SimpleDB je dodávaná jako služba. To znamená, že Amazon neposkytuje možnost nasadit tuto databázi ve vlastním prostředí, ale provozuje ji pouze na svých serverech. Uživatel databáze se tedy nemusí starat o nastavování ani o hardware, ale například ani o škálování, nebo o replikaci dat, jelikož Amazon automaticky distribuuje a synchronizuje data v několika datových centrech umístěných v různých geografických zónách, čímž omezuje nejen riziko ztráty dat, ale také zvyšuje rychlost přístupu k datům v různých místech na světě. Tím pádem se může uživatel databáze plně zaměřit na práci s vlastními daty.

3.2.2 Flexibilita

Flexibilita je další výhoda databáze Amazon SimpleDB. Jelikož je to NoSQL databáze není potřeba mít pevně dané schéma, jak tomu bývá u SQL databází, ale nové atributy a entity mohou být přidávány takzvaně za pochodu.

3.2.3 Indexování

SimpleDB automaticky indexuje veškeré data v ní uložené, čímž tato povinnost odpadá z beder vývojářů aplikace a zvyšuje rychlost navracení výsledků čtecích operací.

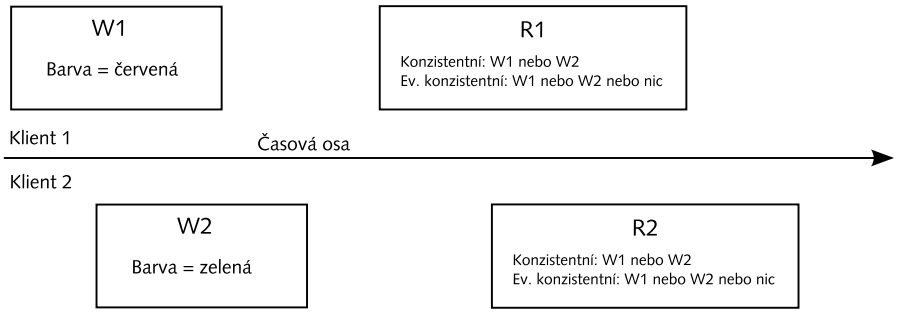
3.2.4 Konzistence

Amazon pro zápis zaručuje konzistenci takovou, že úspěšně provedený zápis (použitím příkazů jako např. *PutAttributes*, *BatchPutAttributes*, *CreateDomain*, ...) znamená, že všechny kopie databáze byly změněny. Toto pojetí konzistence ovšem nezaručuje chování při překrytí dvou či více zápisů najednou. Pouze zaručuje, že výsledné data budou na všech kopiích databáze rovná hodnotě jednoho z těchto zápisů (toho, který databáze obdržela nejpozději). toto bude níže ilustrováno na příkladě. Při čtení si uživatel může zvolit mezi konzistentním a eventuálně konzistentním čtením. Konzistentní čtení zaručuje, že vrátí hodnotu po provedení posledního dosud přijatého zápisu. Eventuálně konzistentní čtení na druhou stranu vrátí hodnotu, která je aktuálně uloená na dotazovaném serveru (tj. může vrátit již neplatnou hodnotu).

Příklad konzistence

Na obrázku 3.1 je uveden příklad, kdy se překrývají dva zápisy. V této situaci není možné určit, jestli pro provedení obou zápisů bude výsledná hodnota atributu `Barva` rovna `červená`, nebo `zelená` z toho důvodu, že není zaručeno, kterou instrukci zpracuje SimpleDB jako první. Například kvůli vysoké latenci sítě prvního klienta bude nejprve přijata a zpracována instrukce `W2` druhého klienta a následně přepsána instrukcí `W1`, tedy veškeré následné konzistentní čtení vrátí `Barva=červená`.

Jak bylo ukázáno, konzistentní čtení v tomto případě mohlo vrátit jak výsledek `W1`, tak výsledek `W2`. Oproti tomu, výsledek eventuálně konzitentního čtení může být kromě `W1` či `W2` také `nic`. Pravděpodobnost prázdného výsledku se s časem minimalizuje, neboli čím déle po zápisu je čteno, tím větší je pravděpodobnost korektního výsledku.



Obrázek 3.1: Ilustrace chování databáze SimpleDB

Základní rozdíl mezi konzistentním a eventuálně konzistentním čtením lze shrnout do tabulky:

Eventuálně konzistentní čtení	Konzistentní čtení
Možné nekorektní výsledky	Vždy korektní výsledek
Rychlejší odezva	Možná pomalejší odezva
Větší dataová propustnost	Možná menší datová prostupnost

Obrázek 3.2: Tabulka vlastností konzistentního a ev. konzistentního čtení

3.2.5 Omezení databáze SimpleDB

Kromě výše zmíněného chování, které by se v jistých situacích dalo považovat za omezení, je asi největším omezením velikost domény, jak co se místa na disku týče, tak také maximální počet atributů. Konkrétně je horní hranice 10 GB pro data a 1 miliarda atributů na jednu doménu. Domén je možno vytvořit až 250 na jeden uživatelský účet.

4 Komunikace s databází Amazon SimpleDB

4.0.6 Přehled příkazů SimpleDB

- **CreateDomain** slouží k vytvoření nové domény.
- **DeleteDomain** vymaže danou doménu.
- **ListDomains** vrátí seznam domén asociovaných s daným uživatelským účtem.
- **DomainMetadata** slouží k získání informací o dané doméně (datum vytvoření, počet položek v doméně, počet atributů, ...)
- **PutAttributes** vloží novou položku s danými hotnotami atributů, či upraví již stávající položku.
- **Select** se podobá běžnému `SELECT` příkazu, který je dobře známý z SQL jazyků. Je ovšem mírně omezen a to například absencí `JOIN` klauzule. Přesný přehled možností příkazu `SELECT` databáze SimpleDB je uveden v dokumentaci[1]
- **GetAttributes** vrátí hodnoty požadovaných atributů jedné dané položky. Umožňuje specifikovat jeden, či více požadovaných atributů, nebo také všechny atributy dané položky.

4.0.7 Podmíněné vkládání a mazání dat

SimpleDB v aktuální verzi (od 24. února 2010) podporuje takzvaný podmíněný zápis a mazání dat.

Při tomto podmíněném zápise (mazání) se spolu s běžnými parametry odešle také podmínka a očekávaný výsledek vyhodnocení dané podmínky. Zápis se poté provede pouze tehdy, je-li vyhodnocená podmínka rovna očekávanému výsledku.

Tato vlastnost se dá použít například pro implementaci čítače (přečte aktuální stav čítače a pokusí se zapsat nový stav čítače za podmínky, že stav čítače byl nezměněn), což by bez podmíněného vložení byl netriviální úkol. Také se tímto rozšiřují možnosti kontroly konkurenčního přístupu k datům (pomocí optimistických protokolů).

---

1. http://awsdocs.s3.amazonaws.com/SDB/latest/sdb-dg.pdf

4.0.8 Bezpečnost

Pro autentizaci má každý uživatel „ID přístupového klíče“ (20 alfanumerických znaků) a „tajný přístupový klíč“ (40 alfanumerických znaků), kterými podepisuje kažý požadavek vůči SimpleDB. Sama Amazon SimpleDB nemá nástroj pro autorizaci, ten je ovšem integrován do správy identit a přístupu AWS (AWS Identity and Access Management - systém pro správu uživatelů a jejich práv v rámci celého AWS), takže je možné omezit práva jednotlivým uživatelům. Jediným problémem může být skutečnost, že nelze definovat přístupové práva pro jiný AWS účet – pro každého uživatele domény musí být vytvořen uživatel v rámci AWS účtu vlastnícího doménu.

Proces autentizace

1. Vytvoření řetězce dotazu
2. Kanonizování řetězce dotazu (seřazení parametrů, zakódování parametrů a jejich hodnot URL kódováním[2]), ...)
3. Vypočítání HMAC[3] z kanonizovaného textového řetězce za použití „tajného přístupového klíče“ jako klíče a SHA256 nebo SHA1 jako hashovacího algoritmu. Tento podpis se připojí k původnímu požadavku jako parametr `signature` spolu s ID přístupového klíče jako `AWSAccessKeyId`.

Zde je ukázán proces autentizace pouze zjednodušeně. Přesný návod jak sestrojit validná požadavek je dostupný v dokumentaci SimpleDB.

4.0.9 Knihovny pro komunikaci

Amazon poskytuje knihovny pro komunikaci se SimpleDB (a ostatními AWS službami) v několika jazycích (Java, PHP, Python, Ruby a .NET). Knihovna pro jazyk Java, která byla v této práci použita, nejenže poskytuje model pro práci s doménami, atributy a podobně, ale také významně usnadňuje autentizační proces (není nutné „ručně“ počítat podpis).

2. Podle RFC 3986, kapitola 2 – http://tools.ietf.org/html/rfc3986
3. Podle RFC 2104 – http://www.ietf.org/rfc/rfc2104.txt

5 Mapování Teiid dotazů na SimpleDB dotazy

V této kapitole bude obecně ukázáno, jakým způsobem byly mapovány Teiid dotazy na dotazy databáze SimpleDB. Konkrétně se jedná o příkazy `SELECT`, `INSERT`, `UPDATE` a `DELETE`.

5.1 Vícehodnotové atributy

Teiid přistupuje k datům jako běžná relační databáze, tedy jednomu řádku a sloupci odpovídá nejvýše jedna hodnota. Z tohoto důvodu je nutné ošetřit přístup k vícehodnotovým atributům. Nabízí se několik řešení:

1. Byl-li by dopředu znám maximální počet hodnot u jednotlivých atributů, mohl by Teiid vnímat vícehodnotové atributy jako více sloupců tabulky (pro každou z možných hodnot jeden sloupec). Tímto přístupem by se ovšem zafixovalo schéma a přišli bychom o jednu z hlavních výhod databáze SimpleDB (SimpleDB netrvá na pevném schématu).

2. Zakódovat všechny hodnoty vícehodnotového atributu do jediného textového řetězce, který by byl lidsky i strojově čitelný. Tímto odpadá problém se změnou počtu hodnot atributu.

Ve finální implementaci byl zvolen druhý přístup a hlouběji bude probrán níže.

5.2 SELECT

Jelikož SimpleDB podporuje mírně omezený `SELECT`, stačí vždy pouze poupravit řetězec příkazu `SELECT` generovaný Teiidem, aby byl srozumitelný pro SimpleDB. Naštěstí lze pro každý překladač nadefinovat sadu podporovaných vlastností. Nepodporované vlastnosti obslouží sám Query Engine (např. překladač nepodporuje `JOIN` klauzuli. Query Engine tedy rozloží dotaz s `JOIN` klauzulí na dva jednoduché dotazy, ty poté předá překladači a výsledné data zpracuje programově v paměti).

Struktura příkazu SELECT databáze SimpleDB:

```
select output_list
from domain_name
where [expression]
[sort_instructions]
limit [limit]
```

Ze struktury je zřejmé, že žádné velké změny nebude třeba provádět. Je nutné definovat, že překladač nepodporuje `JOIN` klauzuli a poté zajistit, aby byly korektně prováděny jednoduché SQL dotazy na které byl původní dotaz rozložen. Zde vyvstává pár problémů:

1. Dotazu obsahujícímu atribut `itemName()` (povinný unikátní atribut, který má každá položka) musí být tento atribut odebrán – tento atribut je automaticky vracen v odpovědi databáze ikdyž není specifikován. Naopak pokud je požadován spolu s jinými atributy, databáze vrátí chybu. Toto platí s výjimkou dotazu, kdy je `itemName()` jediným požadovaným atributem.

   Tento dotaz je korektním dotazem na databázi SimpleDB a vrátí hodnoty atributu `itemName()` všech položek v doméně `TestDomain`:

   ```
   SELECT itemName() FROM TestDomain
   ```

   Tento dotaz vrátí chybovou hlášku `InvalidQueryExpression`:

   ```
   SELECT itemName(), attribute1 FROM TestDomain
   ```

2. Další problém může vyvstat ve výrazu ve `WHERE` části. Těmto potížím se dá vyhnnout správným nastavením schopností překladače.

5.3 INSERT

Zde je situace stále celkem jednoduchá, neboť `INSERT` vždy vkládá jeden nový řádek, což je téměř ekvivalentní příkazu `PutAttributes`. Je zde akorát potřeba získat z Teiid příkazu `INSERT` seznam sloupců, jejich hodnot a název domény do které má proběhnout vložení. Vzhledem ke zvolenému řešení problému vícehodnotových atributů není zde nutné se jimi zaobírat.

5.4 UPDATE

U `UPDATE` je situace poněkud složitější, neboť je možné měnit více položek najednou (díky `WHERE` klauzuli) a příkazy `PutAttributes` a `BatchPutAttributes` neumožňují měnit více položek specifikovaných pomocí kritéria najednou. Je tedy nutné nejprve pomocí `SELECT` příkazu nad požadovanou doménou a kritérii z Teiid `UPDATE` dotazu získat jména všech upravovaných položek (atribut `itemName()`). Teprve poté lze pomocí několika `PutAttributes`, či jediného `BatchPutAttributes` dané položky upravit.

5.5 DELETE

Podobně jako u příkazu `UPDATE` je nejprve potřeba získat jména všech položek, které mají být smazány a ty následně pomocí `DeleteAttributes` smazat z databáze.

6 Implementace

V této kapitole bude rozebrána konkrétní implementace, problémy, které se během implementace vyskytly, a jejich řešení.

6.1 Konektor

Základním kamenem a prvním krokem v implementaci bylo vytvoření konektoru.

Java EE Connector Architecture (JCA) je standardizované řešení pro připojení k podnikovým informačním systémům (enterprise information systems). Stejně jako je JDBC zodpovědné pro připojení Java EE aplikací k databázím, JCA je obecnější architektura pro připojení k systémům, pro které není vytvořen JDBC ovladač. Základem každého konektoru je tzv. resource adapter, jež je odpovědný za přístup a interakci s daným informačním systémem, který cheme řipojit. Jednotlivé Java EE aplikace, nasazené v aplikačním serveru, poté tento resource adapter využívají pro komunikaci.

Specifikace JCA je vyvíjena v rámci Java Community Process jako JSR 322[1]

6.1.1 Modul simpledb-api

Tento modul reprezentuje API pro komunikaci s databází simpledb a jako samostatný modul vzniknul hlavně kvůli oddělení závislostí na java knihovnu pro komunikaci se SimpleDB (aws-java-sdk) od ostatních modulů.

Tento modul obsahuje pouze jednu java třídu a jedno rozhraní a to `SimpleDBAPIClass` a `SimpleDBConnection`.

- `SimpleDBConnection` je rozhraní rozšiřující rozhraní `Connection` z balíčku `javax.resource.cci` (`Connection` reprezentuje připojení na extern zdroj na aplikační vrstvě – díky třídám implementující toto rozhraní je možné přistupovat k požadovaným zdrojům).
- `SimpleDBAPIClass` je třída zaobalující potřebné příkazy databáze SimpleDB. Za povšimnutí stojí hlavně metoda `Set<String> getAttributeNames(String domainName)`, která vrací jména všech atributů v doméně, ovšem jediný způsob jak toho docílit je provedení příkazu `SELECT * FROM <domainName>` a z odpovědi získat jména všech atributů (nestačí dotázat se například na jednu položku, neboť

1. Dostupné na webu Java Community Process – http://www.jcp.org

kdyby daná položka neměla vyplněný nějaký atribut, tento atribut by v odpovědi nebyl obsažen).

6.1.2 Modul connector-simpledb

Jedná se o plnohodnotný, ikdyž poměrně jednoduchý, Java EE konektor pro SimpleDB. Konektory jsou běžně baleny jako RAR (Resource Adapter Archive), které lze poté nasadit na aplikační server pouhým nakopírováním do složky `deploy` serveru. Teiid poskytuje základní implementaci rozhraní potřebných pro vytvoření zákadního konektoru. Zde bylo důležité uzpůsobit tuto základní implementaci potřebám databáze SimpleDB.

- Třída `SimpleDBManagedConnectionFactory` rozšiřuje abstraktní třídu `BasicManagedConnectionFactory` (základní Teiid implementace). V této třídě je nutno definovat proměnné, které jsou potřebné pro připojení ke zdroji – tedy proměnné ve kterých bude uložen ID přístupového klíče a tajný přístupový klíč spolu s metodami pro jejich nastavení a získání (get a set metody). Požadované hodnoty budou aplikačním serverem získány z konfiguračního souboru serveru (typicky třeba `standalone.xml`) a injectovány do těchto proměnných. Další důležitou částí je implementace abstraktní metody `creteConnectionFactory()`, pomocí které je vytvořena instance `ConnectionFactory`. Ta slouží k generování jednotlivých připojení ke zdroji dat (SimpleDB).

- Jako implementace rozhraní `Connection` zde slouží třída `SimpleDB ConnectionImpl`. Ta kromě implementování rozhraní `Connection` rozšiřuje základní Teiid implementaci `BasicConnection`. Nám zde tedy stačí zajistit přístup k instanci třídy `SimpleDBAPIClass`. To je splněno instancováním v konstruktoru a přiřazením do privátní proměnné, která je poté dostupná pomocí její get metody.

  S takto připraveným připojením lze poté v samotném překladači snadno volat metody definované v `SimpleDBAPIClass` např. takto:

  ```
  ((SimpleDBConnectionImpl)connection).getAPIClass().getDomains();
  ```

- Dalším krokem je vytvoření konfiguračních souborů `MANIFEST.MF` a `ra.xml`. Oba soubory je nutno umístit do složky `META-INF`.

  `MANIFEST.MF` obsahuje pouze závislosti resource adapteru. V případě našeho konkrétního adapteru je to:

  ```
  Dependencies: org.jboss.teiid.common-core,org.jboss.teiid.api,
                javax.api,org.jboss.teiid.translator.simpledb.api
  ```

V deployment descriptoru `ra.xml` definujeme hlavní třídu resource adapteru (tj. třída implementující rozhraní `Resource Adapter`), třídu implementující rozhraní `ManagedConnectionFactory` a definujeme proměnné potřebné k připojení k SimpleDB (ID přístupového klíče a tajný klíč).

- Celý tento modul je poté běžně zkompilován a balen jako Resource Adapter Archive – RAR (běžný Java archive – JAR s příponou `.rar`), který může být nasazen na aplikační server zkopírováním do složky `deploy/`. Jelikož se zde ale nejedná o konektor třetí strany, ale o konektor, který bude dodáván s projektem Teiid jako jeden ze základních konektorů, nestačí zajistit, že bude sestaven jako RAR, ale je potřeba, aby ve výsledku byl zařazen mezi moduly aplikačního serveru. K tomu potřebné kroky budou popsány níže v kapitole zabývající se kompilováním a sestavováním projektu pomocí Apache Maven.

6.2 Překladač

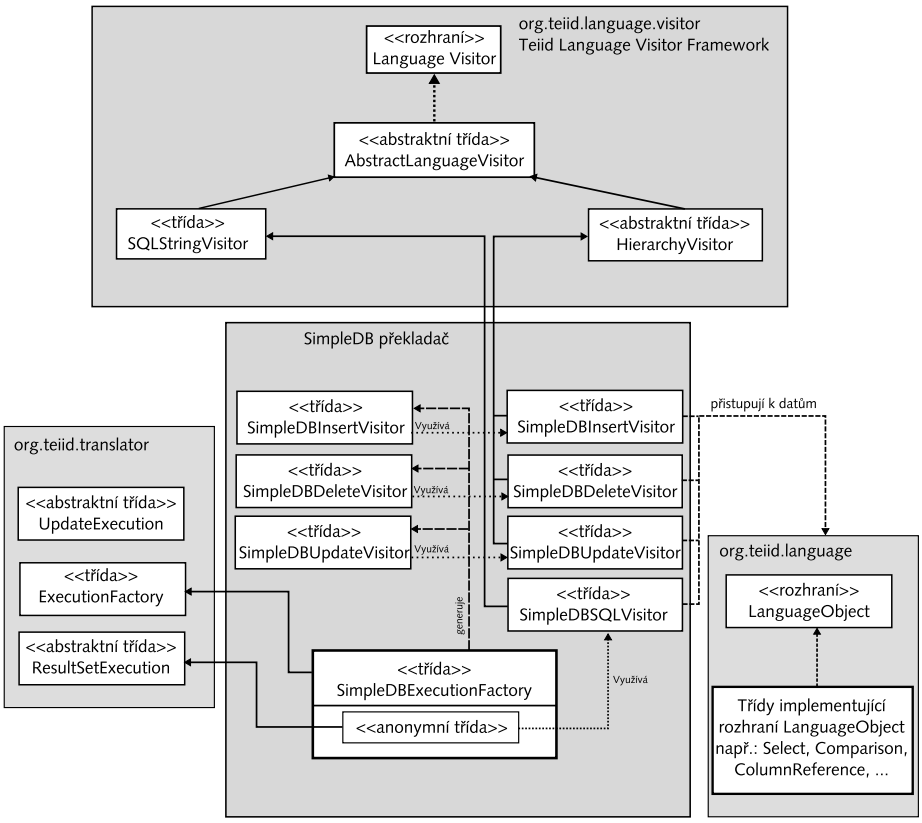
Tento modul je rozdělen na tři logicky oddělené části. V první je pouze třída `SimpleDBExecutionFactory`, která je základní třídou překladače. V další části jsou takzvané executory – třídy zajišťující vykonávání jednotlivých příkazů. Jako poslední jsou takzvané visitory – třídy, které prochází strukturu Teiid SQL příkazu a získávají data, jež jsou poté použita v executorech. Každá z těchto částí je v samostatném java balíčku.

6.2.1 SimpleDBExecutionFactory

Základním kamenem SimpleDB překladače je třída `SimpleDBExecution Factory` rozšiřující abstraktní třídu `ExecutionFactory<F,C>`, kde `F` je factory třída konektoru a `C` je třída připojení – v našem případě se jedná o `ConnectionFactory` a `SimpleDBConnection`.

Nejprve v této třídě najdeme metody definující schopnosti překladače. Jedná se o metody s prefixem `supports` vracející boolean hodnotu podle toho zda překladač danou schopnost podporuje, či ne. Pro překladač SimpleDB byla zvolena tato množina podporovaných funkcí:

```
supportsCompareCriteriaEquals
supportsCompareCriteriaOrdered
supportsInCriteria
supportsIsNullCriteria
supportsRowLimit
supportsNotCriteria
```



Obrázek 6.1: Struktura překladače SimpleDB

`supportsOrCriteria`
`supportsLikeCriteria`
Zbylé funkce obstarává logika Teiidu.

Dále tato třída obsahuje metodu `void getMetadata(MetadataFactory metadataFactory, SimpleDBConnection conn)`, kterou Teiid volá při inicializaci pro každou uživatelem definovanou virtuální databázi. Úkolem této metody je načíst strukturu databáze (tabulky, sloupce, typy, ...) a pomocí `metadataFactory` ji předat Teiidu. Toho je docíleno iterací přes všechny domény a na nich voláním metody `getAttributeNames()` (metoda třídy `SimpleDBAPIClass` k jejíž instanci lze přistupovat přes třídu implementující připojení – `SimpleDBConnection`). Je nutné nastavit všem sloupcům datový typ `String` a také nezapomenout na atribut `itemName()`. Takto získané data poté stačí uložit do instance třídy `MetadataFactory`.

Poslední dvě implementované metody v této třídě jsou metody `create`

```
1  public void getMetadata(MetadataFactory metadataFactory,
2                          SimpleDBConnection conn){
3    List<String> domains = conn.getAPIClass().getDomains();
4    for (String domain : domains) {
5      Table table = metadataFactory.addTable(domain);
6      table.setSupportsUpdate(true);
7      Column itemName = new Column();
8      itemName.setName("itemName()");
9      itemName.setUpdatable(true);
10     itemName.setNullType(NullType.No_Nulls);
11     Map<String, Datatype> datatypes = metadataFactory.getDataTypes();
12     itemName.setDatatype(datatypes.get("String"));
13     table.addColumn(itemName);
14     for (String attrName : conn.getAPIClass().getAttributeNames(domain)) {
15       Column column = new Column();
16       column.setUpdatable(true);
17       column.setName(attrName);
18       column.setNullType(NullType.Nullable);
19       column.setDatatype(datatypes.get("String"));
20       table.addColumn(column);
21     }
22   }
23 }
```

`ResultSetExecution` a `createUpdateExecution`, kde obě mají ve vstupních parametrech příkaz (`SELECT`, `UPDATE`, ...), kontext, metadata a připojení. Obě tyto metody vrací instanci třídy implementující rozhraní `Execution`. Z tohoto rozhraní je pro implementaci překladače pro SimpleDB nejdůležitější metoda `void execute()`, jejíž implementace je odpovědná za provádění konkrétního příkazu.

- `public ResultSetExecution createResultSetExecution(final QueryExpression command, ExecutionContext executionContext, RuntimeMetadata metadata, final SimpleDBConnection connection)`

  Tato metoda je zodpovědná za veškeré příkazy určené k získávání dat – tedy primárně příkaz `SELECT`. `createResultSetExecution` vrací nově vytvořenou anonymní třídu s implementovanými metodami `void execute()` a `List<?> next()`. Metoda `execute` pouze pomocí třídy `SimpleDBSQLVisitor` získá seznam sloupců, poté pomocí stejné třídy získá textový řetězec odpovídající příkazu `SELECT` a nad těmito daty zavolá metodu `performSelect` API třídy, která získá data z databáze a vrátí je jako seznam. Ten je uložen do interní proměnne a v metodě

`next()` jsou tyto data pomocí iterátoru předávána Teiidu.

- `public UpdateExecution createUpdateExecution(final Command command, ExecutionContext executionContext, RuntimeMetadata metadata, final SimpleDBConnection connection)` Zde se obstarává vykonávaní všech metod manipulujících s daty – `INSERT`, `UPDATE`, `DELETE` a BatchedUpdate (který ovšem překladač simpledb nepodporuje).

  Opět, stejně jako u `createResult SetExecution` je zde potřeba vrátit třídu implementující rozhraní `UpdateExecution`, které se od `ResultSet Execution` liší pouze v metodě získávání dat. Místo `List<?> next()` je zde `int[] getUpdateCounts()`.

6.2.2 Executory

Jelikož je zde situace složitější a řešení pomocí anonymní třídy by bylo velmi rozsáhlé, byla zde implementace rozdělena do 3 samostatných tříd podle příkazu, který zpracovávají – `SimpleDBInsertExecute`, `SimpleDBDeleteExecute` a `SimpleDBUpdateExecute`.

Všechny tyto tři třídy využívají visitory (třídy vytvořené podle návrhového vzoru Visitor, budou popsány níže) pro získání relevantních dat ze zadaného příkazu.

`SimpleDBInsertExecute` je nejjednodušší z trojice executorů. Pouze pomocí `SimpleDBInsertVisitor` získá kolekci dvojic `<jménoAtributu, hodnotaAtributu>`, kde hodnota může být ve formátu `^\[.+\]$`, což značí vícehodnotový atribut. Tuto kolekci poté předá metodě `perform Insert()` API třídy. O správné zpracování vícehodnotových atributů se postará tato metoda.

`SimpleDBUpdateExecute` je opět poměrně jednoduchá třída – mezivrstva mezi visitorem a API třídou vykonávající samotný dotaz na SimpleDB. Pomocí visitoru zjistí, které položky odpovídají porovnání v dotazu. Přesněji řečeno získá hodnotu atributu `itemName()` všech položek odpovídajících podmínce `WHERE`. Dále zkonstruuje mapu map `Map<String, Map<String, String>` – pro každý název položky získá pomocí visitoru `SimpleDBUpdate Visitor` mapu `jménoAtributu, hodnotaAtributu>` a s touto mapou map provede pomocí metody `performUpdate()` API třídy samotné upravení dat v databázi.

V případě `SimpleDBDeleteExecute` je zde pro jednoduchost a efektivnost rozděleno vykonávání do tří nezávislých větví podle situací, které mohou nastat:

- Příkaz `DELETE` nemá klauzuli `WHERE`

V tomto případě executor rovnou pomocí API třídy zjistí hodnoty atributu `itemName()` všech položek v dané doméně a tyto položky postupně pomocí metody `performDelete()` API třídy vymaže.

- Příkaz `DELETE` má klauzuli `WHERE` ve formátu `itemName()=<hodnota>`

  Zde se jedná o jednoduché vymazání jediné položky. Tato položka je pomocí metody `performDelete()` vymazána.

- Příkaz `DELETE` má klauzuli `WHERE` ve formátu jiném než `itemName()=<hodnota>`

  V tomto případě je nutné nejprve zjistit hodnoty atributu `itemName()` položek, jež odpovídají podmínce klauzule `WHERE`. Tato logika je ponechána visitoru a zde se poté jen zavolá na získaných jménech metoda `performDelete()` API třídy.

Všechny tyto executory mají povinnost pamatovat si počet upravených, přidaných, či vymazaných řádků a tuto hodnotu zpřístupnit pomocí metody `int[] getUpdateCounts()`

6.2.3 Visitory

Tato sekce obsahuje 4 visitory. Každý ze 4 příkazů `SELECT, UPDATE, DELETE, INSERT` má jiné požadavky, proto pro každý z nich byl vytvořen samostatný visitor.

Nejprve je ale třeba vysvětlit návrhový vzor Visitor, na kterém je celá tato sekce postavená.

Návrhový vzor Visitor

Návrhový vzor visitor (někdy též návštěvník) je jedním ze základních návrhových vzorů popsaných již roku 1994 v knize Design Patterns: Elements of Reusable Object-Oriented Software. Jedná se o návrhový vzor, pomocí kterého je možné přidat skuině tříd dodatečnou funkcionalitu bez nutnosti změnit všechny tyto třídy. Také umožňuje funkcionalitu rozšířit za běhu programu.

V tomto návrhovém vzoru jsou třídy rozdělené do dvou skupin – navštěvované třídy a navštěvující třídy. Navštěvované třídy poskytují pouze základní funkcionalitu a navíc musí implementovat metodu, která příjme návštěvníka a předá mu data nutná k jeho správnému fungování. Návštěvníci pro každou třídu, které chce rozšířit funkcionalitu, implementuje metodu `visit` se vstupním parametrem navštěvované třídy.

TADY MUZU NASYPAT OBRAZEK VISITORU

Jazykový visitor v Teiidu

Teiid disponuje rozhraním pro procházení a získávání dat ze stromů objektů jazyka založené na návrhovém vzoru visitor.

Každý Teiid příkaz je reprezentován stromem jazykových objektů – tříd implementujících rozhraní `LanguageObject`. Toto rozhraní implementuje jedinou metodu:

```
void acceptVisitor(LanguageObjectVisitor visitor);
```

`LanguageObjectVisitor` je základní rozhraní pro všechny visitory a definuje `visit` metody pro všechny jazykové objekty jako například:

```
public void visit(AggregateFunction obj);
```

Teiid kromě základních rozhraní poskytuje i 3 základní implementace, kterých se dá využít při psaní vlastního visitoru:

- `AbstractLanguageVisitor` je prázdnou implementací rozhraní `LanguageObjectVisitor` a poskutující nástroje pro procházení stromu jazykových objektů (sama ale žádné procházení nedělá). Tato třída není určena pro přímé použití, proto je definována jako abstraktní a je určena pouze pro použití jako rodičovská třída.

- `HierarchyVisitor` je podtřídou `AbstractLanguageVisitor` a je oproti její nadtřídě schopna projít všechny uzly stromu jazykových objektů. Nad těmito objekty ovšem neprovádí žádnou akci, pouze poskytuje jakýsi návod jak projít celý strom. Tato třída je také definována jako abstraktní a je určena k rozšiřování.

- `DelegatingHierarchyVisitor` je rozšířením `HierarchyVisitor` o možnost projít strom před a po projití hlavního visitoru. Tento nástroj není v případě SimpleDB nutné využívat, proto se jím nebudeme zabývat dále.

- `SQLStringVisitor` je rozšířením základní abstraktní třídy `AbstractLanguageVisitor`, která dokáže daný objekt Teiid jazyka převést na textový řetězec řetězec odpovídající SQL příkazu.

- `CollectorVisitor` je třída vhodná pro získání všech objektů určitého typu z jazykového stromu.

SimpleDBSQLVisitor

`SimpleDBSQLVisitor` je v této práci použit pouze pro příkaz `SELECT`. Jelikož má ale SimpleDB příkaz `SELECT` mírně odlišnou strukturu a nějaké omezení, nelze použít Teiidem dodávaný `SQLStringVisitor`, ale byla vytvořena tato třída. `SimpleDBSQLVisitor` je rozšířením třídy `SQLStringVisitor` s přetíženýma metodama `visit` pro objekty `Select, ColumnReference, Limit, Like a Comparison`.

- `public void visit(Select obj)`

  Tato metoda je přetížena kvůli nutnosti odstranit atribut `itemName()` ze seznamu sloupců (`itemName()` je vracen automaticky a není třeba se na něj dotazovat)

- `public void visit(ColumnReference obj)`

  Implementace v `SQLStringVisitor` vypisuje sloupec ve formátu `'tabulka' .'sloupec'`, což pro SimpleDB není žádoucí a je potřeba vypisovat sloupec bez tabulky.

- `public void visit(Comparison obj)`

  Jelikož Teiid podporuje operátor `NOT` (a v `SimpleDBExecutionFactory` bylo definováno, že náš překladač operátor `NOT` pdporuje), zdálo by se, že v této oblasti je vše v pořádku. Není tomu ovšem tak, neboť ve chvíli, kdy překladač podporuje `NOT` operátor, zaručuje také podporu negovaných operátorů. Jediný z těchto negovaných operátorů je operátor „nerovná se – <>“, jež musí být při generování SQL řetězce převeden na „NOT =“

```
1 if (obj.getOperator().equals(Operator.NE)){
2   Comparison c = new Comparison(obj.getLeftExpression(),
3                 obj.getRightExpression(), Operator.EQ);
4   append(new Not(c));
5 }
```

- Další změny jsou pouze minoritního charakteru.

SimpleDBInsertVisitor

Vše co je potřeba na vykonání SimpleDB příkazu `PutAttributes` je jméno domény a mapa jména atributu a hodnot atributů. To je zajištěno rozšířením abstraktní třídy `HierarchyVisitor` a přetížením jeho `visit` metod pro parametry typu `ColumnReference` a `ExpressionValueSource` .

Při návštěvě `ColumnReference` (sloupců) se tvoří seznam názvů sloupců. Poté se při návštěvě `ExpressionValueSource` (ekvivalent klauzuli `VALUES`) tvoří mapa, kde jako klíč poslouží jméno atributu a jako hodnota hodnota atributu. Jelikož `HierarchyVisitor` prochází pomyslný strom Teiid příkazu do hloubky, je zajištěno, že seznam jmen atributů bude sestaven před začátkem zpracovávání vkládaných hodnot.

Pro získání těchto hodnot je implementována statická metoda `getColumnsValuesMap(Insert insert)`, která instanciuje vlastní třídu předá ji objektu `Insert` jako návštěvníka, který ji projde. Následně vrátí získané hodnoty.

SimpleDBUpdateVisitor

Ikdyž je i tento visitor založen na `HierarchyVisitor` je zde situace poněkud složitější, než v případě `SimpleDBInsertVisitor` a to hlavně protože pokud v příkazu existuje podmínka, je nutné zjistit jména všech položek odpovídajících podmínce. Proto je zde zvolen jiný přístup, než u `SimpleDBInsertVisitor` (statické třídy pro získávání dat). Zde je navštěvování položek příkazu spuštěno v konstruktoru, tedy při vytvoření instance, a následně lze `get` metodami získat „vydolované“ data.

Konstruktor je parametrický a kromě instance `Update` přijímá i instanci připojení a to z důvodu uvedeného výše.

Požadované data zde jsou: název domény, mapa jmen atributů a jejich hodnot a seznam jmen položek odpovídajících podmínce.

Přetížené metody `visit` jsou pro objekty typu `Update`, `SetClause` a `Comparison`

- `Update` kromě implicitního zavolání metod `visit` pro objekty níže ve stromu příkazu (`SetClause, Condition`) si také uloží jméno domény do privátní proměnné.
- `SetClause` vytvoří a naplní mapu jmen atributů a jejich hodnot.
- Hlavním úkolem metody `visit(Comparison obj)` je získání hodnot atributu `itemName()` všech položek vyhovující podmínce. To lze jednoduše zajistit `SELECT` dotazem přímo na databázi SimpleDB, kde se do `WHERE` klauzule vloží řetězec vygenerovaný pomocí `SimpleDBSQLVisitoru` pro danou podmínku, jak je vidět v následující ukázce kódu na řácích 4-6. Na řádcích 7-8 pak pouze probíhá uložení získaných jmen položek do privátní proměnné `itemNames`.

```
1 public void visit(Comparison obj) {
2   ArrayList<String> columns = new ArrayList<String>();
```

```
3     columns.add("itemName()");
4     List<List<String>> response = apiClass.performSelect("SELECT "
5         +"itemName() FROM "+tableName+" WHERE "+SimpleDBSQLVisitor
6         .getSQLString(obj), columns);
7     for (List<String> list : response) {
8       itemNames.add(list.get(0));
9     }
10  }
```

7 Tutoriál

8 Závěr

Literatura