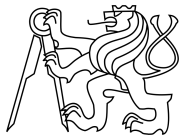
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Studie volby databázové platformy pro realizaci normalizovaných softwarových systémů
Student:	Bc. Lukáš Janeček
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství (magisterský)
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2015/16

Pokyny pro vypracování

1. Seznamte se s teorií normalizovaných softwarových systémů (NS) [1], zaměřte se na požadavky pro data - struktura, persistence, transakční zpracování.
2. Existuje prostředí, které podporuje tvorbu a generování NS na základě specifikace. Přístup k datům je zde řešen technologií EJB. Seznamte se s ním.
3. Proveďte důkladnou analýzu požadavků pro práci s daty z hlediska teorie NS a její implementace pomocí EJB.
4. Seznamte se s principy CAP a BASE, které stojí za návrhy moderních databázových systémů.
5. Vypracujte studii, která zmapuje, jak specifické požadavky na data při návrhu konkrétního NS společně s dalšími požadavky na cílové nasazení (např. vysoká dostupnost, která může vyžadovat distribuci či replikaci dat) ovlivní volbu databázové technologie použitou pro realizaci příslušného NS.

Práce má výzkumný charakter. Hlavním výstupem a přínosem bude studie (viz bod 5.), která bude sloužit jako výchozí bod pro další projekty.

Seznam odborné literatury

[1] H. Mannaert, J. Verelst: Normalized Systems. ISBN 978 90 77 160 008. Koppa Digitale Media. 2009.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 27. listopadu 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Studie volby databázové platformy pro realizaci normalizovaných softwarových systémů

Bc. Lukáš Janeček

Vedoucí práce: Ing. Michal Valenta, Ph.D.

28. dubna 2015

Poděkování

Děkuji svému vedoucímu práce panu Ing. Michalovi Valentovi, Ph.D., za jeho čas a pravidelné konzultace během psaní práce. Dále panu Ing. Robertu Perglovi, Ph.D. a pracovníkům z Normalized Systems Institutu, kteří se mnou komunikovali ohledně teorie NS (zejména Arco Oost a Jan Verelst). V neposlední řadě také děkuji paní Mgr. Janě Divišové za korekturu práce a také své rodině a všem, kteří mě podporovali během psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 28. dubna 2015 .....

Odkaz na tuto práci

Janeček, Lukáš. *Studie volby databázové platformy pro realizaci normalizovaných softwarových systémů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Na začátku práce jsou popsány NoSQL databáze a některé jejich vlastnosti spojené zejména s CAP teorémem. Další kapitola se věnuje teorii NS podle knihy Normalized Systems [15] a jsou zde nejen přesné definice, ale také její intuitivní chápání. Stěžejní částí práce je rozbor jednotlivých NoSQL databází z pohledu jejich vhodnosti pro použití s NS a je zde popsáno existující řešení postavené na technologii EJB.

Klíčová slova Normalizované systémy, Databázové systémy, NoSQL, CAP, BASE, Java EE, JPA

Abstract

There are described NoSQL databases at the beginning of the master thesis and their properties connected with CAP. In the next chapter there is a description of normalized systems based on the book Normalized Systems [15] and its informal understanding. The main part is description of usage NoSQL databases in NS theory and existing implementation based on EJB technology.

Keywords Normalized systems, Database systems, NoSQL, CAP, BASE, Java EE, JPA

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Jedním z velkých problémů dnešních informačních systémů je jejich velikost a složitost. Jedinou jistotou, která u moderních systémů platí, je, že nic není jisté. Požadavky na systém se neustále mění a přibývají nové. To je dáno zejména vysokým tempem rozvoje informačních technologií a vysokou komplexností požadovaných systémů. S tímto jevem se tradiční postupy tvorby softwaru nedokážou příliš dobře vypořádat a výsledné systémy jsou plné chyb nebo nejsou vůbec dokončeny. Pokud jsou tradiční IS provozovány a rozšiřovány po nějakou delší dobu (například 5–10 let), stanou se tak nepřehlednými, že je lepší je naprogramovat zcela znovu, než do nich přidat novou funkcionalitu.

Tomuto problému se snaží zabránit teorie normalizovaných systémů (dále jen NS), která přizpůsobuje architekturu celého systému potenciálním změnám. Jedná se o styl tvorby softwaru, kdy se již dopředu předpokládá, že výsledek se bude neustále měnit, a kdy je vše podřízeno tomu, aby libovolné změny způsobily co nejméně zásahů do stávajícího systému a nezničily jeho přehlednost.

Dalším trendem dnešní doby je obrovské množství informací a potřeba po jejich uložení a zpracování. Generování a ukládání velkého množství dat se často nazývá BigData a díky jejich existenci vznikají nové možnosti a vědní obory. Mají také obrovský vliv na spoustu odvětví, včetně managementu firem, reklamy, data miningu, porozumění trendům a dalších. Tyto trendy si však žádají nejen nový přístup k informacím, ale i nové technologie a způsoby práce s nimi. Jednou z hlavních technologií, která se díky BigData ohromně rozrůstá a zcela mění, jsou databáze. Tradiční relační databáze nebyly schopny zpracovávat takové množství dat, a proto vznikají takzvané NoSQL databáze přizpůsobené těmto potřebám. BigData jsou důvodem vzniku mnoha NoSQL databází a jejich postupného prosazování v komerčních produktech.

Dalo by se říci, že trendem dneška je mít data a umět z nich vytěžit potřebné informace. Zcela stěžejní je tato schopnost pro firmy a jejich vedení. Ideálem jsou tedy systémy, které nejenže pracují s velkými daty a jejichž

údržba a rozšiřitelnost téměř nic nestojí, ale které dokážou velmi rychle reagovat na požadované změny. Často se totiž stává, že zapracování nové funkcionality do systému trvá velmi dlouho (pokud se to vůbec podaří) a firma tak ztrácí schopnost agilně reagovat na změny. Právě těmto problémům by mohlo zabránit použitní NoSQL databází spolu s teorií normalizovaných systémů.

Tato práce se zabývá možnostmi použití existující technologie normalizovaných systémů s NoSQL databázemi. Je velice důležité si uvědomit, jak a proč NoSQL databáze fungují a jaké mají vlastnosti. První kapitola proto popisuje základy teorie různých druhů databází a vlastnosti jako CAP a BASE. Jsou zde také vyjmenováni někteří zástupci různých skupin databází a jejich vlastnosti. Další kapitola se věnuje teorii NS, jejich formálnímu i intuitivnímu popisu a snaží se zavést českou terminologii. Součástí je také srovnání NS s existujícími paradigmaty v programování. Jelikož je existující implementace NS spojena s technologií Java EE a specifikací EJB, jsou zde také stručně popsány tyto technologie. Zbytek práce se pak věnuje samotnému použití NoSQL databází v NS a rozebírá způsoby, jak mapovat data, jak vytvářet datové schéma nebo kterou databázi zvolit.

Kapitola 1

Cíl práce

Cílem práce je zmapování různých systémů, převážně z řad NoSQL databázových strojů, které se v dnešní době používají k ukládání dat, z hlediska vhodnosti pro použití v systémech, navržených podle pravidel a předpokladů plynoucích z teorie normalizovaných systémů [15]. První a neméně důležitou částí práce bude analýza požadavků na data plynoucích z této teorie a také základní přehled vlastností a možností nabízených jednotlivými databázovými stroji.

Nejprve se zaměříme na několik vybraných NoSQL databázových strojů a popíšeme jejich vlastnosti a nejčastější případy použití. Zaměříme se zejména na typ ukládaných dat (strukturovaná/nestrukturovaná data), možnosti jejich zpracování (ukládání, editaci a vyhledávání) a také na možnosti transakčního zpracování. Důležitou charakteristikou je podpora určitých požadavků z CAP teorému popřípadě z vlastností ACID. Většina NoSQL systémů totiž není ACID, ale pouze BASE a zde by mohly vzniknout největší problémy s použitím těchto systémů pro účely NS. Požadavky ACID, BASE a CAP budou vysvětleny v následující kapitole.

Kapitola 2

NoSQL databáze a jejich vlastnosti

Nejprve zmíníme vlastnosti, které jsou kladeny na SQL databáze a zmíníme způsob jejich splnění v praxi. Poté rozebereme, kterých vlastností obyčejně dosahují NoSQL databáze a v čem jsou odlišné.

2.1 ACID

Všechny SQL databáze musí podle standardu SQL [21] splňovat požadavky ACID a úroveň serializable.

- **A**tomicity (atomicita) – Atomicita znamená, že provedené změny se provedou jako jeden nedělitelný celek. Tedy provedou se veškeré změny, nebo žádné. Aby bylo jasné, které změny se mají provést atomicky, používá se takzvaná databázová transakce (posloupnost příkazů, u kterých je zaručena atomicita). Pokud se transakci nepodaří provést, oznámí to SŘDB pomocí chybového hlášení.

- **C**onsistency (konzistence) – Pojem konzistence se vztahuje na stav databáze během a po skončení jedné transakce. Splněním této vlastnosti je zaručeno, že se po provedení transakce nedostane systém do nekonzistentního stavu, tedy že jsou splněna všechna integritní omezení. Jinými slovy, že transakce převádí databázi z jednoho konzistentního stavu do druhého.

- **I**solation (izolovanost) – Izolovanost zaručuje, že data, která jsou měněna v rámci transakce, uvidí ostatní transakce až po jejím úspěšném ukončení (příkazu COMMIT).

- **D**urability (trvalost) – Trvalost zaručí, že veškeré změny, které proběhly v rámci úspěšně ukončené transakce, jsou trvale uloženy a nemohou být ztraceny.

Některé z těchto vlastností jsou však často velice přísné (izolovanost) a jsou na úkor rychlosti celého systému. Uvolněním některé z nich však může dojít k některým z transakčních anomálií. Naopak při dodržení všech požadavků může ve víceuživatelských databázích docházet k uváznutí (deadlock). Databáze tedy často implementují různé úrovně izolace, které zabraňují jednotlivým transakčním anomáliím:

- Dirty read (špinavé čtení) – Z databáze jsou načtena data změněná v jiné transakci, která však ještě nebyla potvrzena (příkazem COMMIT).
- Nonrepeatable read (neopakovatelné čtení) – Dvě po sobě jdoucí čtení stejných dat vrátí stejný počet záznamů, ale budou obsahovat jiné hodnoty.
- Phantom reads (výskyt fantomů) – Dvě stejná čtení vrací jiný počet záznamů.

Pokud dvě a více transakcí (klientů) přistupuje paralelně (ve stejný čas) ke stejným datům, může docházet ke konfliktům. Dojde-li ke konfliktu, mohou se objevit takzvané transakční anomálie. Které anomálie se mohou objevit, určují stupně izolace. Jejich přehled je v tabulce 2.1.

	špinavé čtení	neopakovatelné čtení	výskyt fantomů
read uncommitted	ano	ano	ano
read committed	ne	ano	ano
repeatable read	ne	ne	ano
serializable	ne	ne	ne

Tabulka 2.1: Stupně izolace [21]

Pouze poslední izolační stupeň splňuje podmínku izolovanosti, degraduje však přístup k databázi téměř na sériový. Z tohoto důvodu většina databázových strojů pracuje standartně pouze v režimu read commited [21].

2.2 BASE

Pomyslnou náhradou za přísný přístup ACID je v jistém smyslu uvolněný BASE přístup.

- **B**asically **A**vailable (dostupný) – Aplikace pracuje bez přerušení a data jsou v podstatě dostupná.

- **S**oft state (částečně konzistentní) – Databáze nemusí být v každém okamžiku zcela konzistentní.
- **E**ventual consistency (eventuelně konzistentní) – Databáze se v určité době stane konzistentní.

Hlavní rozdíly obou přístupů podle Brewera [1] jsou znázorněny v tabulce 2.2.

ACID	BASE
silná konzistence	slabá konzistence (stará data)
izolovanost	dostupnost na prvním místě
orientace na commit	přibližné odpovědi jsou OK
vnořené transakce	jednodušší, rychlejší
dostupnost	dodávka dat „jak jen to půjde“
konzervativní (pesimistické)	agresivní (optimistické)
složitá evoluce (schema...)	jednodušší evoluce

Tabulka 2.2: Srovnání vlastností ACID a BASE [1]

2.3 CAP

CAP teorém je jakýsi teoretický rámec vlastnostem BASE vzhledem k tomu, že vlastnost izolovanosti se často porušovala. CAP platí pro silně distribuované systémy. Tento teorém má pouze tři požadavky.

- **C**onsistency (konzistence) – Všichni klienti (všechny uzly distribuovaného systému) vidí ve stejný čas stejná data.
- **A**vailability (dostupnost) – Každý klient může vždy číst i zapisovat data.
- **P**artitioning tolerance (odolnost vůči rozdělení) – Systém funguje, i když vypadne některý z uzlů, nebo se rozpadne na více nezávislých částí.

Jak je již z jednotlivých bodů vidět, jedná se zejména o vlastnosti při práci v distribuovaných systémech. Problém ale je, že nikdy nemohou být splněny všechny tři vlastnosti naráz. To dokázal Eric Brewer [1]. Po několika letech rozvoje databází se ale ukazuje, že CAP nemusí být bráno jako dogma. Například Eric Brewer ve svém článku CAP teorém po 12 letech [2] uvádí několik poznámek ohledně používání batabází v praxi.

Vztah ACID a CAP je často chápán špatně, protože písmena C a A jsou v obou, ale vždy znamenají něco jiného. Nejprve je nutné si uvědomit, že měření metrik C, A i P je relativní a nelze snadno určit, zda daná vlastnost platí, nebo ne. Podívejme se na jednotlivé vlastnosti z pohledu zbylých dvou.

- Pokud povolíme jednomu uzlu změnit data, ostatní budou dočasně nekonzistentní, takže nesplňujeme konzistenci.
- Pokud zachováme konzistenci, část uzlů musí být nedostupná, dokud se data nezpropagují, takže nesplňujeme dostupnost.
- Pouze pokud uzly komunikují, může být zachována konzistence i dostupnost, pak ale ztrácíme odolnost vůči rozdělení.

U velkých systémů nemůžeme v praxi ztratit toleranci vůči rozdělení (P), a proto se musíme rozhodnout mezi dostupností (A) a konzistencí (C). NoSQL databáze se obvykle přiklání k dostupnosti. Možnost splnění pouze dvou vlastností ze tří je zavádějící a nemusí vadit vždy:

1. Víceuzlové (distribuované) databáze jsou vzácné.
2. Výběr mezi dostupností a konzistencí může být zvolen podle určité operace nebo specifických dat a nemusí být pevný pro celou aplikaci.
3. Všechny tři vlastnosti jsou jemnější než pouze ano/ne. Například hodnoty dostupnosti jsou od 0 do 100%, také je více úrovní konzistence a některé uzly nemusí být vždy brány v potaz.

Jelikož je tedy distribuovanost vzácná, můžeme si dovolit perfektní dostupnost i konzistenci. Pokud ale máme distribuovanost, můžeme postupovat podle následujících kroků:

1. Rozpoznat rozdělení.
2. Vstoupit do speciálního „rozděleného“ módu a omezit některé operace.
3. Po opětovném spojení uzlů zahájit proces na obnovení konzistence a opravu chyb vzniklých během rozdělení.

Jaké operace omezit? E. Brewer [2] uvádí příklad, kdy požadavek na unikátnost klíčů se obvykle rozhodneme riskovat a povolíme duplicitní klíče během rozdělení. Při obnovení je duplicitní klíče lehké detekovat a předpokládáme-li, že tyto hodnoty mohou být sloučeny, můžeme obnovit konzistenci. Naopak převod peněz z jednoho účtu na jiný provést nemůžeme. Nahraje se tady pouze požadavek, který se provede po obnovení celé sítě. Z toho plyne, že po obnovení musí být stav na obou stranách konzistentní a musí existovat mechanismus, jak opravit chyby, které vznikly během rozdělení. Často se vyjde z posledního konzistentního stavu a provádějí se postupně nahrané změny. Při použití komutativních operací se předchází problémům při obnovování. Další možnosti jsou:

- Poslední změna (časově) vyhrává.

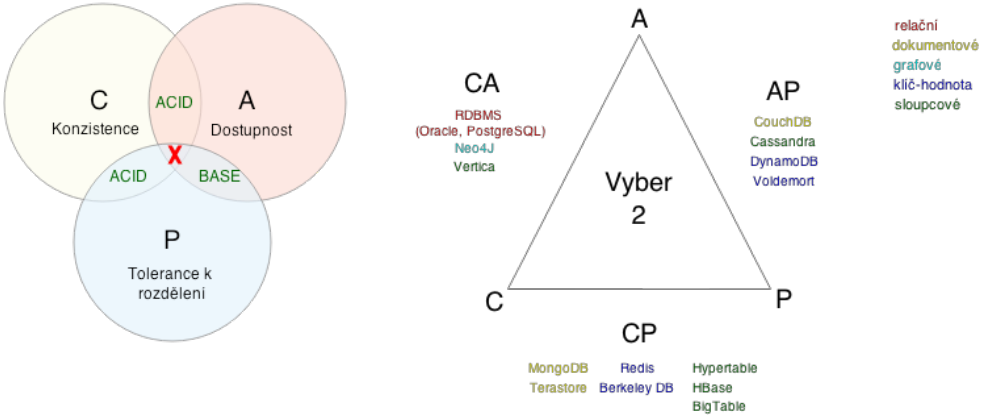
- Odložit nebezpečná rozhodnutí až na konzistentní stav.
- Někdy není možné přímo opravit chyby, ale můžeme je nějak kompenzovat.

CAP teorém může také velice ovlivnit latence mezi jednotlivými uzly. Po uplynutí timeoutu se musí rozhodnout:

1. zrušit operaci ⇒ snížit dostupnost
2. provést operaci ⇒ riskovat nekonzistenci

Pokus o novou komunikaci pouze opozdí toto rozhodnutí. Z výskytu latence plyne, že některé uzly mohou detekovat rozdělení a jiné ne. Někdy tedy můžeme uvolnit konzistenci a spravovat vzdálené kopie asynchronně. Pokud mají uživatelé službu nedostupnou, mohou někdy pracovat v offline módu. V tomto případě si volíme dostupnost místo konzistence a musíme se zotavovat z dlouhého rozpojení uzlů. Například můžeme zaručit dostupnost a konzistenci na jednom uzlu, zatímco na jiném je služba nedostupná.

Z tohoto přehledu je vidět, že pohled na CAP teorém není tak černobílý, jak by se mohlo na první pohled zdát, a že v praxi ho lze použít i bez větších omezení celého systému.



Obrázek 2.1: Znázornění CAP teorému

2.4 NoSQL databáze

Co to tedy jsou NoSQL databáze? NoSQL není obecně chápáno jako: „žádné SQL“, ale spíše se tímto pojmem rozumí „not only SQL“, tedy: „ne pouze SQL“. Podle [17] se NoSQL databáze vyznačují následujícími charakteristikami:

- bez relačního datového modelu (bez schématu)
- distribuované
- open-source
- horizontálně škálovatelné
- snadná replikace (podpora distribuované architektury)
- jednoduché API
- BASE (ne ACID)
- schopnost pracovat s velkým množstvím dat

Jelikož těchto databází existuje veliké množství, často se člení ještě do kategorií podle způsobu ukládání dat a práce s nimi. V následující části se podíváme na některé z nich a několik jejich zástupců. Je ale potřeba upozornit, že některé databáze mohou spadat do více skupin najednou.

Ještě před samotným popisem jednotlivých databází popíšeme přístup Map / Reduce, který se v NoSQL databázích velice často používá pro přístup k datům.

2.4.1 Map / Reduce

MapReduce je způsob, jak zpracovávat velké objemy dat pomocí paralelního zpracování na více uzlech. Tento přístup má dvě fáze:

1. **Map**, kdy vezme řídící uzel vstupní data a rozdělí je na menší části ostatním uzlům ke zpracování. Všechny uzly zpracují svá data nezávisle na ostatních a výsledky předají zpět řídícímu uzlu.
2. **Reduce**, kdy řídící uzel převezme vypočtená data od zpracovatelů, odstraní duplicity, provede jejich agregaci a vrátí výsledek.

Fáze Map může být i víceúrovňová, kdy zpracovatel opět rozdělí obdržená data a předá je dalším zpracovatelům, pro které se stane řídícím uzlem.

2.4.2 Databáze klíč-hodnota (Key-value)

Key-value je jednoduchý model, kdy jednomu adresovatelnému unikátnímu klíči odpovídá právě jedna hodnota. Tato hodnota může být jak strukturovaná, tak i nestrukturovaná, ale databáze „nevidí dovnitř“ a nerozumí datům v hodnotách. Tyto databáze jsou velice jednoduché, a proto jsou velice flexibilní a dobře škálovatelné. Nevýhodou je, že veškerá logika musí být přenesena na stranu aplikace.

2.4.2.1 Redis

Redis je open-source databáze, ve které lze ukládat následující struktury [8]:

- String – jakýkoli druh dat až do velikosti 512MB
- Hash – asociativní pole
- List – seznam řetězců seřazený podle pořadí vložení
- Set – neseřazená množina unikátních řetězců (podporuje mnoho množinových operací)
- SortedSet – seřazený set

Data jsou uchovávána v operační paměti a pravidelně ukládána na pevný disk. Redis používá master-slave replikaci, kdy je slave přesnou kopií mastera a zápis je povolen pouze na master uzel. Redis podporuje transakční zpracování [8].

2.4.3 Sloupcové NoSQL databáze (Column-oriented)

Sloupcové databáze (Wide Column Store / Column Families) ukládají data do tabulek adresovatelných pomocí indexu řádku. Na rozdíl od relačních databází může mít každý řádek jiný počet sloupců a je možné sloupce přidávat. Každý sloupec má své unikátní jméno, hodnotu pro daný řádek a časové razítko.

2.4.3.1 Cassandra

Cassandra vychází z Google BigTable a Amazon DynamoDB a byla původně vyvíjena Facebookem. Obsahuje takzvané column family (CF), která sdružují data stejného typu (podobně jako tabulka v relačních databázích). Cassandra podporuje i super sloupce, tedy sloupce, jejichž hodnotou je mapa jiných sloupců. Cassandra se vyznačuje následujícími vlastnostmi [27].

- Lineárně škálovatelná.
- Podporuje Map / Reduce.
- Data jsou řazena v okamžiku zápisu.
- Každý sloupec obsahuje kromě hodnoty také časové razítko.
- Vlastní dotazovací jazyk CQL.

2.4.4 Dokumentové databáze

Dokumentově orientované databáze umožňují ukládat komplexnější data, takzvané dokumenty. Dokumentem se rozumí nějak strukturovaná data (typicky ve formátu JSON [22]). Jedná se vlastně o databáze klíč-hodnota, kdy ale hodnota je strukturovaná a databázový stroj „vidí dovnitř“. Každý dokument jednoznačně identifikuje jeho klíč. Tyto databáze podporují sekundární indexování uvnitř dokumentu.

2.4.4.1 MongoDB

MongoDB je open-source databáze, která má následující vlastnosti:

- Podporuje horizontální škálování (automatické dělení dat do více instancí a ukládání souvisejících dat pohromadě).
- Dokumenty jsou seskupovány do kolekcí (podobně jako tabulka v relačních databázích), které mohou být heterogenní.
- Dokumenty jsou ukládány ve formátu BSON (binární podoba JSONu) a indexovány pomocí _id hodnoty.
- Má vlastní dotazovací jazyk podobný SQL a podporuje Map / Reduce (napsaný pomocí JavaScriptu).
- Je možné vytvářet indexy pro rychlejší vyhledávání.

2.4.4.2 CouchDB

Apache CouchDB je distribuovaná databáze s následujícími vlastnostmi:

- Má RESTful HTTP/JSON API.
- Data nelze mazat, pouze se přidávají nové revize s jiným časovým razítkem.
- Podporuje ACID transakce a dotazy pomocí Map / Reduce.
- Umožňuje obousměrnou replikaci a offline provoz včetně následného obnovení.

2.4.5 Grafové databáze

Grafové databáze se skládají z uzlů (vrcholů) a hran. Vrcholy představují objekty a hrany jejich vazby (vztahy). Hrany i vrcholy mohou mít atributy.

2.4.5.1 Neo4j

- Podporuje ACID vlastnosti [8].
- Obsahuje webové rozhraní s grafickým znázorněním dat.
- Obsahuje vlastní dotazovací jazyk CYPHER, nativní Java API nebo RESTové rozhraní přístupné přes HTTP.
- Každá vazba má typ, může být orientovaná a obsahovat atributy (dvojice klíč-hodnota).
- Uzly mají také typ a skládají se z dvojic klíč-hodnota.

2.4.6 Objektové databáze

Objektové databáze mají rozhraní pro objektové jazyky a umožňují ukládat přímo celé hierarchie objektů. Z pohledu programátorů se chovají podobně jako ORM (objektově-relační mapování). Objektových databází se lze dotazovat jednak pomocí speciálního jazyka OQL, ale také nativně, přímo v daném programovacím jazyce. Každý objekt je identifikován jednoznačným OID, což je ukazatel do virtuální paměti. Není tedy potřeba pro objekty vytvářet primární klíče. Mezi zástupce patří například databáze GemStone, EyeDB nebo ObjectDB.

2.4.7 XML databáze

XML databáze ukládají data v podobě XML dokumentů. Tyto databáze mají několik výhod. Jednak těží z toho, že XML je rozšířená technologie a existuje na ni mnoho nástrojů. Může se jednat například o definici struktury pomocí XML Schémat nebo DTD, vyhledávání pomocí XPath nebo transformací pomocí jazyka XSLT. Jedná se o lidsky čitelný formát, a proto mu je lépe rozumět. V neposlední řadě také existuje speciální dotazovací jazyk XQuery, který spojuje možnosti SQL a XPath výrazů pro dotazování nad daty. XQuery má FLOWR strukturu [11]:

- FOR – výběr posloupnosti uzlů k dalšímu zpracování
- LET – přiřazení proměnných pro každý prvek posloupnosti
- WHERE – filtrování uzlů v posloupnosti
- ORDER BY – seřazení vybraných uzlů
- RETURN – specifikace výstupu

Mezi zástupce patří například eXist, dbXML, XHive/DB, BaseX nebo Sedna. Tyto databáze však mají velice specifické použití a moc často se nevyskytují v aplikacích s paralelním přístupem více uživatelů. Z tohoto důvodu jim nebude v dalších kapitolách věnován moc velký prostor.

2.4.8 Shrnutí NoSQL databází

Podle Bena Scofielda [23] se jednotlivé skupiny databází vyznačují vlastnostmi uvedenými v tabulce 2.3.

Model	Výkonnost	Škálovatelnost	Flexibilita	Složitost	Funkčnost
Klíč-hodnota	vysoká	vysoká	vysoká	minimální	proměnná
Sloupcové	vysoká	vysoká	průměrná	malá	malá
Dokumentové	vysoká	proměnná	vysoká	malá	proměnná
Grafové	proměnná	proměnná	vysoká	vysoká	-
Relační	proměnná	proměnná	malá	průměrná	-

Tabulka 2.3: Nabízené vlastnosti jednotlivými skupinami databází podle [23]

Jak je tedy vidět, každá skupina NoSQL databází nabízí jiné vlastnosti a dokonce i jednotliví zástupci daných skupin se od sebe mohou lišit. Z obrázku 2.1 je vidět, že i databáze mající stejný datový model mohou splňovat odlišné vlastnosti z CAP teorému. Pro výběr správné databáze pro konkrétní aplikaci nebo její část je důležité zvážit nejen samotné datové schéma, ale také požadavky na rychlost, dostupnost, distribuovanost, replikaci,... Díky velkému množství databází (na stránkách `http://nosql-database.org` [15] jich je aktuálně uvedeno 150) je však velice pravděpodobné, že se odpovídající databáze podaří najít.

Nejvhodnějším řešením však bývá kombinace vícero databází v rámci jedné aplikace. Tomuto řešení se říká polyglot persistence a více o něm je zmíněno v kapitole 6.2.

Kapitola 3

Teorie normalizovaných systémů

Teorie normalizovaných systémů (NS) se zabývá konstrukcí informačních systémů, které se vyznačují dobrou schopností reagovat na změny a adaptovat se. Tato teorie vznikla na univerzitě v Antwerpách na fakultě aplikované ekonomiky. Později univerzita ustanovila Normalized Systems Institute pro další výzkum a vývoj této teorie [10]. NS jsou založeny na výzkumu Herwiga Mannaerta a Jana Verelsta. Tato teorie je naprosto nezávislá na použité technologii nebo frameworku, ale existuje referenční implementace, napsaná pomocí technologie Java EE.

V následující kapitole se pokusím vymezit základní pojmy definované v knize Normalizované systémy [15] a základní požadavky na jednotlivé stavební bloky.

3.1 Základní principy

Většina problémů spojená s implementací informačních systémů je spojená s jejich komplexností a s potřebou jejich neustálého rozvoje a změn. Přidání nové funkcionality může generovat (a v praxi to tak často opravdu je) takzvané kombinatorické efekty[1]. To vede k přílišné komplexnosti a zvyšující se náročnosti (potažmo ceně) dalších změn. Tyto myšlenky formuloval Manny Lehman následovně [14]:

> A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.

Program, který se používá v reálném prostředí a který se neustále mění, aby odrážel realitu, se stává méně užitečným až do doby, kdy vyjde levněji nahradit systém zcela novou verzí.

[1] Jejich dopad není závislý pouze na velikosti změny. Více v kapitole 3.2.

> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

Volně přeloženo to znamená, že při evolučních změnách je program stále méně strukturovaný a vzrůstá jeho vnitřní složitost. Odstranění složitosti vyžaduje dodatečné úsilí.

V roce 1968 popsal Doug McIlroy vizi budoucího vývoje informačních systémů, ve kterých budou systémy sestavovány místo toho, aby byly programovány [16]:

> Expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able safely to regard components as black boxes.

Podle této vize bude možné jednotlivé díly systémů přepoužívat bez potřeby znát jejich vnitřní strukturu.

Teorie NS předpokládá, že změna systémů je jejich neoddělitelnou součástí a musí se s ní počítat již při počátečním návrhu a vývoji systému. Tato teorie definuje sadu pravidel, které umožňují implementovat nezávislé komponenty, ze kterých jsou skládány samotné systémy.

3.2 Základní model informačních systémů

Aby se dala definovat potřebná pravidla pro normalizované systémy, musíme si nejdříve vyjasnit některé pojmy.

- *Stabilita (stability):* omezená množina změn vede k omezené množině dopadů.
- *Předpoklad neomezené evoluce systému (PNES) (assumption of unlimited systems evolution):* předpoklad, že počet změn půjde do nekonečna.
- *Kombinatorické efekty (combinatorial effects):* změna, pro kterou platí, že velikost jejího dopadu není závislá pouze na charakteru změny, ale také na velikosti celého systému.
- *Normalizované systémy (normalized systems):* systémy, které vykazují / zaručují stabilitu vzhledem k očekávané množině změn.
- *Primitivum (primitive):* datová nebo akční entita (Definice 5 a 6) nebo úloha (task) (Definice 7).

Základní předpoklad, který vystihuje cíl teorie NS je podle [15] následující:

Předpoklad 1. *Informační systém musí být stabilní vzhledem k definované množině předpokládaných změn.*

Tento předpoklad se snaží zabránit Lehmanovu výroku o neudržovatelnosti informačních systémů při jejich neustálém vývoji a zamezit kombinatorickým efektům spojeným se změnami funkčnosti. Pro dosažení tohoto předpokladu je nutné definovat základní model informačních systémů, na kterém zavedeme pravidla, pomocí nichž dosáhneme stability.

Definice 1. *Softwarový modul je část systému nebo programu, který má následující vlastnosti:*

- *modul má rozhraní obsahující:*
  - *jméno modulu, které popisuje funkce a zodpovědnost modulu a které je použito k volání modulu*
  - *vstupní parametry modulu*
  - *výstupní a návratové parametry modulu*
- *modul obsahuje seznam příkazů obsahujících:*
  - *data (jako: proměnné, konstanty, datové struktury)*
  - *instrukce (jako: základní instrukce, funkce, procedury)*

Definice 2. *Normalizované systémy jsou informační systémy, které jsou stabilní vzhledem k definované množině očekávaných změn, což vyžaduje, aby omezená množina těchto změn vyústila v omezenou množinu dopadů na primitiva systému.*

Nyní definujeme několik konstruktů, ze kterých se složí základní model informačních systémů.

Definice 3. *Softwarovou entitou rozumíme instanci (např. funkci) v konkrétním technickém prostředí.*

Definice 4. *Základní model informačního systému, nezávislý na konkrétní technologii, může být vyjádřen následujícími primitivy:*

- *datové entity (Data entities)*
- *akční entity (Action entities) obsahující úlohy (tasks)*

Definice 5. *Datová entita (Data entity) je softwarová jednotka obsahující různé atributy nebo hodnoty, včetně odkazů na další datové entity.*

Datová entita (jako je struktura nebo record (např. v C++)), nemá interface, tedy žádné rozhraní k vnějšímu světu.

Definice 6. *Akční entita (Action entity) je softwarová jednotka, která reprezentuje operaci. Akční entita obsahuje jeden nebo více úloh (tasks).*

Akční entita bere datové entity jako vstup a produkuje je jako výstup.

Definice 7. *Úloha (task) je součást akční entity a jedná se o množinu instrukcí, které provádějí určitou funkcionalitu. Úloha je zařazena na submodulární úrovni.*

Definice 8. *Externí technologie (external technology) v určité úloze reprezentuje jednu nebo více softwarových jednotek (entit), které náleží k jinému technologickému prostředí, ať se jedná o jiný programovací jazyk, framework, softwarový balík a/nebo knihovnu potenciálně doménově specifickou.*

Nyní již můžeme definovat základní model informačních systémů. Je možné si ho představit jako složeninu výše definovaných elementů následovně:

1. Základní pohled na systémy: IS je množina akcí, které zpracovávají data.
2. Model IS může být vyjádřen následujícími primitivy (definice 4):
    - datové entity – obsahuje atributy nebo hodnoty (definice 5).
    - akční entity – reprezentuje operace s daty, obsahuje jednu nebo více úloh (definice 6).
        - úloha – sada instrukcí provádějící určitou funkcionalitu (definice 7). Úloha může používat externí technologii, framework, funkci, ...
            * externí technologie – přítomnost SW entity, která náleží externímu prostředí (definice 8).

Z předpokladu 1 a definice 2 vyplývá, že musíme definovat množinu předpokládaných (nebo očekávaných) změn, které mohou v libovolném systému nastat a které pokrývají veškeré možné změny, které můžeme při dalším vývoji požadovat.

Definice 9. *Pro základní model informačního systému definujeme následující množinu očekávaných změn:*

- *Dodatečný atribut nebo hodnota*
- *Dodatečná datová entita*
- *Dodatečná akční entita, včetně možností jako:*
    - *Dodatečná akční entita, která má specifickou datovou entitu jako vstup nebo produkuje specifickou datovou entitu jako výstup.*
    - *Dodatečná akční entita volající specifickou akční entitu*
- *Dodatečná verze úlohy, včetně možností jako:*

- *Dodatečná verze využívající jiné externí technologie*
- *Dodatečná verze představující povinnou aktualizaci (upgrade)*
- *Dodatečná verze obsahující nový chybový stav*

Jak je vidět, uvažujeme pouze změny spojené s přidáním nějaké položky či elementu. Jak se tedy vypořádat s mazáním nebo modifikací? Tyto dvě změny můžeme bez omezení na obecnosti brát následovně:

- modifikace – kombinace přidání a smazání primitiva
- smazání – není změna, pouze záležitost GC (garbage collectoru[2])

Nyní máme již definována všechna potřebná primitiva pro libovolný informační systém a můžeme tedy vyslovit čtyři základní věty, které tvoří základ NS.

3.2.1 Oddělení zodpovědnosti (Separation of Concerns)

Oddělení zodpovědnosti znamená rozdělení programu na více částí podle funkcionality. Každá část je zodpovědná za určitou funkcionalitu. Příkladem může být n-vrstvá architektura (MVC), SOA nebo integrační bus (ESB).

Věta 1. *Akční entita může v normalizovaných systémech obsahovat pouze jednu úlohu (task).*

Důkaz 1. *Nechť N je počet akčních entit E_i, které kombinují úlohu A s různými verzemi úlohy B. Zavedení nové povinné verze úlohy A povede k N změnám v akčních entitách E_i. Podle PNES může být N neomezené $\Rightarrow$ počet změn bude neomezený.*

Důsledky:

→ Každá akční entita v NS může obsahovat pouze jednu externí technologii.

→ Pokud je workflow implementováno v externí technologii nebo podpůrné technologii, musí být odděleno do nové úlohy.

→ Implementace průřezových problémů (crosscutting concerns) představuje oddělenou úlohu. Tedy podpůrné úlohy vyžadují oddělenou akční entitu.

Průřezové problémy (crosscutting concerns) jsou problémy, které se týkají více částí aplikace. Může se například jednat o přihlašování, zpracování transakcí, validace dat, ukládání dat, zabezpečení, logování, profilování.

[2]Mechanismus, který se stará o odstranění instancí, které již v IS nejsou použity.

3.2.2 Transparentnost datové verze (Data version transparency)

TDV (transparentnost datové verze) znamená, že datové entity mohou mít více verzí, aniž by se to dotklo akcí, které s nimi pracují ⇒ přidání hodnoty neovlivní akční entity, které s touto hodnotou nepracují.

Věta 2. *Datová entita, která je přijímána jako vstup nebo produkována jako výstup některou akční entitou, musí mít v normalizovaných systémech transparentní verzi.*

Důkaz 2. *Nechť N je počet akčním entit E_i, které dostávají na vstup specifickou datovou entitu D. Pokud D nevystavuje transparentní verzi, přidáním hodnoty do D povede k N změnám ve všech entitách E_i. Podle PNES může být N neomezené ⇒ počet změn bude neomezený.*

Důsledky:

→ Datové entity, které jsou předávány jako parametry akčním entitám, musí skrývat (zapouzdřovat) svoje data.

→ Průřezové problémy datových entit mohou být přidány jako zvláštní akční entita.

Například v OOP můžeme TDV dosáhnout používáním bezparametrického konstruktoru a metod get- a set- pro přístup k datům.

3.2.3 Transparentnost akční verze (Action version transparency)

Úlohy mohou mít více verzí, jelikož akční entita obsahuje pouze jednu úlohu (podle věty 1) ⇒ akční entita může mít více verzí. Transparentnost akční verze znamená, že akce (akční entity) mohou být změněny (vyměněny za jinou), aniž by se to dotklo akcí, které je volají.

Věta 3. *Akční entita, která je volána jinou akční entitou, musí v normalizovaných systémech vystavovat transparentní verzi.*

Důkaz 3. *Nechť N je počet akčních entit E_i, které volají specifickou akční entitu A. Pokud A nevystavuje transparentní verzi, přidáním nové povinné verze A povede k N změnám ve všech entitách E_i. Podle PNES může být N neomezené ⇒ počet změn bude neomezený.*

Důsledky:

→ Samostatná akční entita musí obalovat akční entity reprezentující verzi úlohy.

→ Průřezové problémy akčních entit mohou být přidány jako samostatná akční entita a měly by být implementovány na úrovni obalující akční entity.

Implementace může být například pomocí obalující funkce nebo pomocí polymorfismu.

3.2.4 Udržování stavu (Separation of States)

Udržování stavu vyžaduje, že jednotlivé akce volané v rámci nějakého workflow si udržují stav, který je aktualizován po volání každé z nich. To vede k použití asynchroního a stateful volání.

Věta 4. *Volání akční entity jinou akční entitou si musí v normalizovaných systémech udržovat stav.*

Důkaz 4. *Nechť N je počet akčních entit E_i, které volají specifickou akční entitu A. Předpokládejme, že není udržován žádný stav, dokud se neukončí akční entita A. Zavedením nové verze A, která může obsahovat nový chybový stav, povede k N změnám všech akčních entit E_i tak, aby zachytávaly tento chybový stav. Podle PNES může být N neomezené ⇒ počet změn bude neomezený.*

Důsledky:

→ Stav akční entity musí být udržován pro každé volání akční entity ⇒ stav musí být součástí nebo spojen s datovou entitou, která je předána jako vstup akční entity. Přítomnost konkrétního stavu v datové entitě může spustit zavolání akční entity pro tento stav.

→ Je možné, ale ne nutné, aby byl stav uchováván perzistentně. Při pádu se tak neztratí data. Je ale nutné, aby mechanismus persistence nebyl zajišťován akční entitou.

→ Udržování stavu umožňuje zachycování výjimek (chyb) ne pomocí vracení výjimky v hierarchii volání, ale pomocí uložení chyby a umožnění jiné akční entitě na tuto chybu reagovat.

3.3 Pokročilý model informačního systému

Aby bylo možné zajistit, že se všechny předpokládané změny provedou bez kombinatorických efektů, musíme definovat nový pokročilý model informačního systému, složený z nových zapouzdřených elementů. Nejprve definujeme, co je to návrhový vzor.

Definice 10. *Návrhový vzor (design pattern) popisuje strukturované složení a komunikaci mezi softwarovými entitami.*

Definice 11. *Pokročilý model informačního systému, nezávislý na nějaké technologii, může být vyjádřen pomocí následujících primitiv:*

- *Datové elementy*
- *Akční elementy*
- *Workflow elementy*
- *Konektor elementy*
- *Trigger elementy*

Jak je vidět v definici, pokročilý model informačního systému se skládá z pěti vyšších elementů. Tyto elementy nyní definujeme jako návrhové vzory. Pro jejich definici budeme potřebovat ještě dvě pomocné definice 12 a 13.

Definice 12. *Funkční úloha (function task) je úloha provádějící konkrétní funkci v informačním systému.*

Definice 13. *Podpůrná úloha (supporting task) je obecná úloha provádějící podpůrné (průřezové) úlohy v informačních systémech.*

3.3.1 Zapouzdření dat (data encapsulation)

Zapouzdření datových entit do datového elementu požaduje:

- Podle věty 2 mají datové elementy get- a set- metody pro zaručení transparentnosti datové verze v širším slova smyslu (pro komunikaci v rámci aplikace) a metody marshal- a parse- v užším slova smyslu (pro serializaci a deserializaci, například při použití ve webových službách).
- Podpůrné úlohy a průřezové problémy mohou být přidány v souladu s větami 1 a 2.

Definice 14. *Datový element (data element) reprezentuje množinu atributů nebo hodnot (fields), včetně odkazů na jiné datové elementy, a může obsahovat podpůrné úlohy. Interně je strukturován jako návrhový vzor složený ze softwarových entit.*

3.3.2 Zapouzdření akcí (action encapsulation)

Zapouzdření akčních entit do akčních elementů požaduje:

- Podle věty 1 musí každá akční entita obsahovat pouze jednu funkci nebo úlohu.
- Podle věty 2 musí být argumenty a parametry zapouzdřené datové entity.

- Podle vět 1 a 4 musí být workflow odděleno od akčních entit (a zapouzdřeno ve workflow elementech).
- Podle věty 3 musí být úlohy zapouzdřeny tak, že samostatná akční entita obaluje akční entity reprezentující verze úlohy.
- Podpůrné úlohy (průřezové problémy) mohou být přidány v souladu s větami 1 a 3.

Definice 15. *Akční element (action element) reprezentuje jednu zapouzdřenou funkční úlohu na modulární úrovni a může obsahovat podpůrné úlohy. Interně je strukturován jako návrhový vzor složený ze softwarových entit.*

Akční element musí podporovat a obalovat více verzí jeho funkční úlohy. Tomu říkáme, že akční element může mít více implementací (každá nabízí stejnou funkcionalitu).

Definice 16. *Implementace akčního elementu je softwarová entita obsahující verzi funkční úlohy.*

3.3.3 Zapouzdření workflow (workflow encapsulation)

Složení softwarových entit do workflow elementů požaduje:

- Podle věty 1 nemohou workflow elementy obsahovat žádné jiné úlohy.
- Podle věty 4 musí udržovat stav (be stateful). Stav je vyžadován pro každou instanci akčního elementu a musí být svázán s datovým elementem, který je předáván jako argument.

Definice 17. *Workflow element reprezentuje sekvenci akčních elementů, které jsou vykonávány s uchováváním stavu. Interně je strukturován jako návrhový vzor složený ze softwarových entit.*

3.3.4 Zapouzdření konektoru (connector encapsulation)

Složení softwarových entit do konektor elementů požaduje:

- Podle věty 4 se musí konektory ujistit, že externí systémy mohou pracovat s datovými elementy, ale nemohou volat akční elementy bez udržování stavu.
- Podpůrné úlohy mohou být přidány v souladu s větami 1 a 3.

Definice 18. *Konektor element (connector element) reprezentuje vstupní a výstupní úlohu (IO) v informačním systému. To zahrnuje jak člověk-počítač IO, například grafické uživatelské rozhraní, tak také počítač-počítač IO s externí aplikací nebo zařízením. Interně je strukturován jako návrhový vzor složený ze softwarových entit.*

3.3.5 Zapouzdření spouštěčů (trigger encapsulation)

Složení softwarových entit do trigger elementů požaduje:

- Podle věty 4 trigger elementy musí odděleně kontrolovat chybové i nechybové stavy a kontrolovat, kdy mají být spuštěny akční elementy.
- Podpůrné úlohy mohou být přidány v souladu s větami 1 a 3.

Definice 19. *Trigger element reprezentuje aktivaci akčního nebo workflow elementu na základě nějaké periody, jedná se tady o časově závislý element. Interně je strukturován jako návrhový vzor složený ze softwarových entit.*

3.3.6 Předpokládané změny v pokročilém modelu IS

S rozšířeným modelem IS musíme také definovat rozšířenou sadu předpokládaných změn. Tyto změny pokrývají veškeré zásahy, které můžeme v informačním systému provést.

Definice 20. *Pro pokročilý model informačního systému definujeme následující množinu předpokládaných změn:*

- *Dodatečný atribut*
- *Dodatečný datový element*
- *Dodatečný akční element, včetně možností jako:*
  - *akční element mající specifický datový element jako vstup nebo produkující specifický datový element jako výstup*
- *Dodatečná verze funkční úlohy akčního elementu nebo nějaké podpůrné úlohy, včetně možností jako:*
  - *použití dodatečné externí technologie*
  - *povinná aktualizace verze (upgrade)*
- *Dodatečná akce ve workflow elementu*
- *Dodatečný workflow element*
- *Dodatečný konektor element*
- *Dodatečný trigger element*

Tyto změny umožňují přidat libovolné primitivum a jejich počet může být neomezený.

3.4 Neformální popis NS

Normalizované systémy můžeme chápat jako soubor složený z mnoha prvků několika málo druhů. Základními prvky jsou *datové elementy* a *akční elementy*. Jak již název napovídá, datové elementy obsahují data (pouze data, ať už se jedná o primitivní datové typy, nebo reference na jiné objekty) a akční elementy obsahují akce (podle Věty 1 pouze jednu akci), tedy funkčnost. Aby bylo možné data i akce měnit, je potřeba, aby byly po změně použitelné stejným způsobem jako předtím. Jelikož *teorie NS povoluje pouze přidání*, odstraňování se může provádět pouze v případě, že se daný prvek již v celém systému nevyskytuje (tedy odstranění není součástí dané změny, ale pouze jakýmsi pročištěním systému). Aby bylo možné provádět změny (přidávat prvky – jejich výčet je v Definici 9), je nutné zapouzdření jak funkcí, tak dat. O tom jsou Věty 2 a 3.

Zapouzdření dat můžeme dosáhnout použitím *bezparametrického* konstruktoru (Kdyby konstruktor obsahoval nějaké parametry, při přidání nového parametru by se muselo jeho volání změnit na všech místech, kde byl použit.) a metod *get−* a *set−*. Zapouzdření akcí lze implementovat jako rozhraní (interface), které je používáno ve všech entitách a přidání nové třídy, která implementuje toto rozhraní, neovlivní žádné entity. Místo rozhraní lze také použít obalující funkci, která vždy převolává aktuální implementaci akce. Vstupem i výstupem akčních elementů musí být vždy zapouzdřené datové elementy (Při změně dat se díky tomu nebude muset měnit akce.).

Jelikož akční elementy mohou obsahovat pouze jednu úlohu (funkci), musely vzniknout další elementy, které by implementovaly nějaký proces nebo posloupnost akcí. Těmi jsou *workflow elementy*. Workflow elementy tedy určují, jak se budou volat akční entity, a podle Věty 4 musí udržovat stav o jejich průběhu. Jedná se tedy o mechanismus, kdy je vyvolána akční entita, která provede nějakou operaci, a výsledek této operace je uložen (může být i persistentně). Výsledkem může být jak očekávaný výstup, tak i chyba programu. Pokud se vyskytne chyba, musíme na ni nějak zareagovat.

Pokud by se chyba šířila systémem, jako je to v případě výjimek v Javě, bylo by při přidání nové potenciální chyby (chybového stavu) nějakého elementu vždy potřeba upravit všechny elementy, které s ním pracují. Díky udržování stavu workflow a jeho uložení můžeme nechat speciální elementy – *trigger elementy*, aby na ni reagovaly. Tudíž při přidání nového chybového stavu stačí pouze přidat nový element ošetřující tento stav.

Posledním druhem elementů jsou *konektor elementy*. Ty se starají o komunikaci s okolním světem, jako je například člověk (user interface) nebo vzdálený počítač.

Stávající implementace NS je v technologii Java EE a funguje následujícím způsobem: Každá aplikace má sadu konfiguračních souborů (popis konfiguračního souboru pro datové elementy je uveden v kapitole 6.1) a z těchto

souborů se několika skripty generuje výsledná aplikace. Tato aplikace obsahuje velké množství tříd a po jejich vygenerování by měla být zcela funkční. Existují skripty, které generují celý nový element, ale také skripty, které do již existujícího elementu pouze přidávají nové prvky. Aby se tímto přidáváním nepřepsaly programátorovy úpravy, jsou do vygenerovaných souborů vkládana návěstí (v podobě komentářů), která ohraničují vygenerované sekce a mimo ně lze kód libovolně upravovat.

Kapitola 4

Způsoby implementace NS

Nyní popíšeme NS z pohledu existujících paradigmat a programovacích technik, jak je zmiňují ve své knize Jan Verelst a Herwig Mannaert [15]. Jak již plyne z předchozí kapitoly, pro úspěšnou implementaci je minimálně potřeba jazyk s podporou funkcí a datových struktur. Dále pak také mechanismus, který dokáže funkce zapouzdřit v souladu s větou 3.

4.1 Objektově orientované programování

Objektově orientované programování všechny tyto požadavky splňuje, ale umožňuje, ba dokonce programátory nabádá k porušování základních předpokladů NS, a to zejména důrazem na zabalení dat a funkcí do jednoho konstruktu – třídy.

Třídy tak často reprezentují nejen data, ale také obsahují metody. Typicky obsahují průřezové problémy a to vede ke kombinatorickým efektům. Dalším prohřeškem je zabalení metod, které provádějí konceptuálně rozdílné akce, do jedné třídy. Děje se tak často proto, že tyto metody pracují se stejnými daty, nebo proto, že pro jednu metodu nechceme dělat jinou třídu. Dalším prohřeškem je řetězení volání metod, které porušuje udržování stavu. Problémem také je, že metody mohou přímo přistupovat ke všem členům třídy a nemohou být proto použity jako akce, které reprezentuje pouze jejich interface. To vede k silné provázanosti, jež není vidět na první pohled.

Dalším problémem může být dědění. To opět vede k silné provázanosti jednotlivých datových atributů a také metod. Zděděné metody mohou obsahovat závislosti na externí technologie a jejich volání může vést k řetězovému převolávání mnoha akcí. Podle [15] vede použití již pětiúrovňové dědičnosti k problémům a podle předpokladu PNES jde tedy o nebezpečný konstrukt.

Ani návrhové vzory nejsou řešením. Porože OO třídy mohou být nízkoúrovňové a nestrukturované, nezdá se být jako vhodné řešení aplikovat na ně předdefinované struktury v podobě návrhových vzorů. Dalším problémem je vysoká provázanost a komplexnost stávajících návrhových vzorů.

Pokud se ale vyvarujeme výše popsaných konstruktů, může OOP sloužit jako poměrně schopný prostředek pro potřeby NS. Třídy mohou sloužit pro obalení dat a metod (zvlášť) podle vět 2 a 3. Třídy pak zaručují transparentnost datové verze použitím get- a set- metod a transparentnost akční verze pomocí různých tříd se stejným interfacem.

4.2 Aspektově orientované programování

Častým problémem v OOP je opakování kódu na více místech. Tento kód je obvykle spojen s průřezovými problémy, jako je přihlašování, zabezpečení, transakční zpracování,... Aspektově orientované programování nám umožní oddělit tento kód od business kódu aplikace. AOP upravuje chování na přesně definovaných místech (joinpoint) pomocí vložení nového kódu (advice). Tyto advice se pomocí speciálních výrazů (pointcut) aplikují na konkrétní joinpoint a upravují chování nebo parametry dané metody, nebo dokonce zamezí jejímu spuštění [9].

Samotná teorie NS s tímto postupem souhlasí, ale podle [15] je zbytečné zavádět další konstrukty do jazyka. Stejného cíle lze dosáhnout použitím klasických tříd a metod a navíc lze tyto konstrukty často generovat. Navíc apekty mohou opět přinést vyšší provázanost mezi komponentami.

4.3 Servisně orientované programování

Používání servis se vyznačuje malou provázaností, což je dobře. Také zapouzdření funkčnosti v rámci servis vede k transparentnosti akčních verzí a použití XML jako přenosového formátu implikuje zapouzdření datových verzí. Díky XML reprezentujícímu data nemůže vzniknout zapouzdření dat s funkcemi. Problémem SOA je velký důraz na workflow, které je založeno na jazyku BPEL. Toto řešení jednak neudržuje stav, ale také neodděluje zodpovědnost, a to vede ke kombinatorickým efektům.

Kapitola 5

Použití Javy EE v teorii NS

Java EE (Enterprise Edition) je platforma pro vývoj podnikových aplikací. Jedná se o sadu specifikací, které prošly dlouhým vývojem a mají usnadnit vývoj rozsáhlých systémů sadou doporučení a podpůrnými nástroji, zejména pro průřezové a často se opakující problémy. Na této technologii je postaveno referenční řešení NS z Normalized Systems Institutu, a proto v následující kapitole představíme základní principy a mechanismy této technologie.

Součástí platformy Java EE jsou především specifikace pro [26]:

- vývoj webových aplikací – JavaServer Pages (JSP), JavaServer Faces (JSF)
- vkládání závislostí – dependency injection
- přístup k relačním databázím – Java Persistence API (JPA)
- vývoj sdílené business logiky – Enterprise Java Beans (EJB)
- přístup k frontám zpráv – Java Messaging Service (JMS)
- podporu webových služeb – Java API for XML Web Services (JAX-WS), Java API for RESTful Web Services (JAX-RS)
- podporu transakcí – Java Transaction API (JTA)

Aby mohly být tyto funkcionality používány, musí aplikace běžet ve speciálním prostředí, takzvaném aplikačním serveru (AS). Těchto serverů je mnoho, například GlassFish, JBoss JOnAS, Apache Tomcat, IBM WebSphere, WebLogic,... Tyto servery se starají například o komunikaci s klienty, životní cyklus komponent, správu databázových spojení, transakční zpracování, persistenci objektů, asynchronní komunikaci,...

Nebudeme se nyní zabývat příliš detailně celou Javou EE (tématika je příliš rozsáhlá), ale zaměříme se pouze na základní koncepty pro pochopení této technologie tak, jak je použita v referenční implementaci NS.

Java Bean je znovupoužitelná softwarová komponenta, která zapouzdřuje další objekty, je serializovatelná a obsahuje bezparametrický konstruktor a gettry a settry pro všechny vlastnosti (property). Z těchto vlastností vychází koncept Enterprise Java Beans (EJB).

Session Beans jsou beany, zapouzdřující business logiku aplikace a mohou být volány klientem. Existují beany, uchovávající stav konverzace s klientem (stateful), bezestavové (stateless), které po ukončení metody veškerý stav zahodí, a jedináčci (singleton), tedy beany, pro které existuje v celém kontejneru pouze jedna instance. Přístup k beanám je pomocí rozhraní (interface):

- lokálních (local) – definuje metody, které mohou volat komponenty ve stejném kontejneru
- vzdálených (remote) – definuje metody, které mohou být volány ze vzdálených kontejnerů

Message Driven Beans jsou beany, které se nevolají přímo, ale jsou zavolány kontejnerem po příchodu zprávy.

EJB kontejner je prostředí, do kterého se nasazují EJB komponenty. Stará se zejména o komunikaci se vzdáleným klientem, dependency injection, správu stavů, životní cyklus komponent, transakce, ...

Deployment descriptor je XML soubor, který popisuje, jak má být modul nebo aplikace nasazena na webový server.

5.1 Java persistence API

Java persistence API (JPA) je standard popisující rozhraní knihoven pro objektově-relační mapování. Jelikož se jedná pouze o specifikaci, existuje pro ni více implementací. Nejznámější z nich jsou například Hibernate, EclipseLink, OpenJPA nebo ObjectDB.

Entity Bean (nebo také entita) je třída, pro kterou kontejner obstarává objektově-relační mapování. Pro Entitu musí platit [26]:

- musí obsahovat anotaci javax.persistent.Entity
- musí mít public nebo protected konstruktor bez parametrů
- persistentní proměnné nesmí být public a klient k ním nesmí přistupovat přímo
- třída ani její metody nesmí být $final$
- třída musí obsahovat atribut s anotací javax.persistence.Id

- měla by být serializovatelná (implementovat rozhraní Serializable)

Mezi entitami a databází se nachází takzvaný *persistentní kontext* [7]. Jedná se o množinu instancí, které jsou spravovány jedním entityManagerem (viz dále). V jednom persistenčním kontextu je nejvýše jedna instance entity daného typu se stejným primárním klíčem (jedná se tedy o jakousi cache). Jelikož stav entit v persistenčním kontextu nemusí odpovídat stavu databáze, musí dojít k takzvané synchronizaci (na konci transakce při volání commit nebo zavoláním metody $flush()$). Dojde-li k chybě (např. porušení integritního omezení), je v databázi volán rollback, ale entity se do původního stavu nevrací.

Entity v persistentním kontextu jsou takzvaně spravované (managed), mimo něj jsou odpojené (detached).

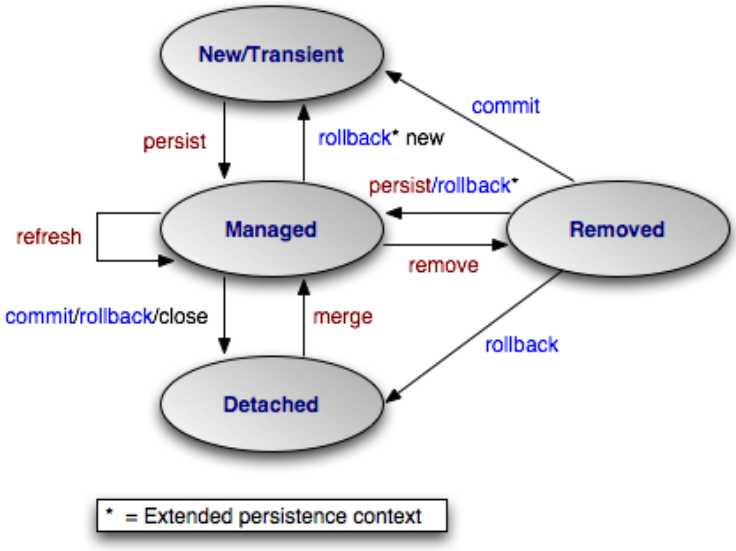
Aby bylo možné entity ukládat do nějaké databáze, musí se vytvořit *persistentní jednotka* (konfigurace se nachází v souboru $persistence.xml$). Ta definuje DataSource (údaje o databázovém serveru, jako umístění, jméno, heslo, ...), konkrétní implementaci JPA, která bude použita, a také entity, které budou touto jednotkou spravovány. Samotné operace s entitami (ukládání, vyhledávání, mazání,...) obstarává *EntityManager*. Ten se také stará o cachování, transakční služby a životní cyklus entit. Jeho rozhraní musí obsahovat následující metody [7]:

- *find* – načte entitu se zadaným klíčem z úložiště do kontextu (operace SELECT)
- *refresh* – obnoví entitu v kontextu podle úložiště
- *persist* – přidá entitu do kontextu (po peraci $flush()$ se entita uloží do databáze) (operace INSERT)
- *merge* – upraví entitu v úložišti dle jejího stavu v kontextu (operace UPDATE)
- *remove* – odebere entitu z úložiště (operace DELETE)
- *detach* – odebere entitu z kontextu

Tyto metody umožňují přesuny entit mezi databází a persistentním kontextem. Jejich znázornění je na obrázku 5.1[3].

Standard JPA také definuje dotazovací jazyk JPQL (Java Persistence Query Language), který se podobá jazyku SQL. JPQL je ale nezávislý na použité databázi, místo jmen tabulek a sloupců se používají jména tříd a jejich atributů a JPQL respektuje jejich vztahy.

[3]Rozšířený persistentní kontext může přežít více transakcí. Klasický (ohraničený transakcí) persistentní kontext existuje stejně dlouho jako transakce a zaniká s jejím dokončením.



Obrázek 5.1: Životní cyklus entit [5]

Pro nastavení jednotlivých parametrů mapování se nejčastěji používají anotace. Seznam některých z nich je uveden v tabulce 5.1 [18].

Anotace označující vztah (jsou označeny *) mohou obsahovat následující parametry:

- *cascade* – definuje operace, které se kaskádově provádí na asociovaných objektech
  - ALL – kaskáduje všechny operace
  - MERGE – kaskáduje operaci merge
  - PERSIST – kaskáduje operaci persist
  - REFRESH – kaskáduje operaci refresh
  - REMOVE – kaskáduje operaci remove
- *fetch* – označuje „líné“ dotahování odkazovaných objektů
  - LAZY – dotáhnou se, až když jsou potřeba
  - EAGER – načtou se při přístupu na odkazující element
- *mappedBy* – v případě obousměrné vazby definuje podřízený element
- *optional* – určuje, zda je možné mít daný objekt nedefinovaný
- *targetEntity* – označuje odkazovanou třídu (při použití interface)

Anotace	Umístění	Popis
@Entity	třída	Deklaruje entitní třídu
@Table(name)	třída	Specifikuje jméno tabulky
@Embedded	atribut	Uloží odkazovanou třídu (označenou anotací @Embeddable) přímo do aktuální tabulky
@Column(name)	atribut	Specifikuje název sloupce atributu
@Id	atribut	Označuje primární klíč objektu
@GeneratedValue	atribut s @Id	Přenechá generování na databázi
@Transient	atribut	Atribut, který není persistentní
@UniqueConstraint	atribut	Položka musí být unikátní
@Version	atribut	Atribut představující verzi objektu (pro optimistické zamykání)
@NamedQueries	třída	Deklaruje seznam pojmenovaných JPQL dotazů
@NamedQuery(name,query)	třída	Konkrétní pojmenovaný JPQL dotaz
@ManyToMany(*)	atribut	Vazba m:n
@ManyToOne(*)	atribut	Vazba m:1
@OneToMany(*)	atribut	Vazba 1:n
@OneToOne(*)	atribut	Vazba 1:1
@JoinColumn(name)	atribut	Název sloupce s cizím klíčem
@JoinTable	atribut	Určuje vztahovou tabulku vazby

Tabulka 5.1: Popis anotací ze specifikace JPA.

Existují dva druhy vazeb:

- Jednosměrné (unidirectional) – pouze jedna instance má referenci na druhý objekt.
- Obousměrné (bidirectional) – instance má referenci na druhý objekt a ten má referenci nazpět.

5.2 EJB a NS

Teorie EJB je velice podobná teorii NS. Používají se zde Entity Beany pro data a Session Beany pro akce. Tyto beany jsou velice podobné akčním a datovým elementům. Zapouzdřují tedy data a akce a nabízejí standardní mechanismy

pro mnoho průřezových problémů jako persistenci (pomocí ORM) nebo vzdálený přístup. Ale ani přístup Javy EE nepodporuje všechny požadavky NS. Například následující požadavky zde nejsou podchyceny [15]:

- Nejsou zde modely pro kontrolu workflow ani pro aplikační konektory.
- Mechanismus persistence v EJB není dostatečně silný na to, aby vždy zajistil stabilní přístup k datovým elementům[4].
- Není zde jasný rozdíl mezi implementací podpůrných úloh persistence a vzdáleného přístupu v EJB Entity Beanách.
- Není zde standardní mechanismus pro použití EJB komponent bez znalosti EJB technologie.

Aby mohly být implementovány jednotlivé elementy pokročilého modelu IS v technologii EJB, byly vytvořeny takzvané expandery (generátory), které na základě vstupní specifikace daného elementu vygenerují sadu tříd a zajistí všechny potřebné vlastnosti elementu. Popis expanze a vnitřní struktura vygenerovaných elementů je v příloze B.

[4]Nevynucuje všechny požadavky NS.

Kapitola 6

Požadavky NS na práci s daty

V této kapitole popíšeme požadavky, které jsou kladeny na datové elementy za účelem jejich persistence. Tyto požadavky mohou být jednak na strukturu samotných dat, ale také na průběh jejich zpracování. Z celé teorie NS se nyní omezíme pouze na datové elementy, protože jako jediné reprezentují data informačních systémů, a tudíž vyžadují persistenci.

Jak již ze samotné teorie, ale také z referenčního řešení plyne, persistence objektů je průřezový problém. Tyto úlohy se tedy vyskytují napříč celou aplikací a musí se o ně starat společný mechanismus. Jak sami autoři teorie NS uvádějí [15], pokud zvolíme nějakou knihovnu, komponentu nebo framework, který se stará o takovéto problémy, pak jeho výměna za jiný (s potenciálně jiným API) povede vždy ke kombinatorickým efektům. Této vlastnosti nelze zabránit. Vezměme tedy v úvahu nějaký mechanismus mapování objektů do libovolné NoSQL databáze.

Tento mechanismus bude pro NS vystupovat jako externí komponenta obstarávající průřezové problémy. Není však nikde řečeno, že o persistenci se musí starat pouze tento mechanismus a že nemůže být kombinován například s klasickým ORM (zajišťovaným například aplikačními servery podle specifikace JPA). Tato kombinace však není možná v rámci jedné úlohy. Můžeme tedy brát práci s nerelačními databázemi jako další z mnoha podpůrných úloh, které zajišťuje platforma a aplikace je používá pomocí volání jejího API.

Samotné datové elementy jsou popsány svým jménem a atributy. Každý atribut má nějaké jméno a typ. Tento typ může být jiný datový element nebo primitivní datový typ. Primitivní datové typy jsou předurčeny použitým jazykem (obvykle znak, celé číslo, číslo s plovoucí desetinnou čárkou, boolean a někdy také řetězec). Datový element může dále obsahovat podpůrné úlohy [12]. Tyto úlohy jsou často společné pro mnoho aplikací a lze je tedy znovupoužívat. Některé (celkem 10 [12]) z podpůrných úloh jsou součástí expanderu vyvíjeného na univerztě v Antwerpách a mohou být tedy součástí každé vygenerované aplikace. Jednou z těchto úloh je také persistence.

Podpora NoSQL databází může být tedy zavedena pouze úpravou/náhra-

dou stávajícího expanderu datových elementů, ale může být také naprosto nezávislá na procesu expanze a pracovat nad stejnou strukturou dat, kterou je schopen vytvořit existující expander. V následující práci zkusíme zvážit obě varianty.

6.1 Požadavky na datovou strukturu

Požadavky na strukturu dat a na způsob jejich ukládání a zpracovávání (ve smyslu vyhledávání, spojování,...) jsou dány jednak samotnou teorií, ale také požadavky na vstupní data pro jednotlivé expandery, potažmo jejich výstup. Proto zde uvedeme syntaxi popisných souborů datových elementů podle Element Expanders Developers Guide (verze 0.95) [12]. Příklad vstupního souboru *Purchase.dd* pro expander je v kódu 6.1:

```
1  /home/mannaert/cvs-palver/
2  sitenet net.palver.uams Purchase tsms
3  String name ynn
4  String company yyn
5  String productCode yny10_20_30_40_50
6  String status yyyInitial_Processing_Finished
7  Integer amount ynn
8  Ln01net.palver.site.Site atSite nnn
9  Ln01net.palver.site.Technician orderedBy nnn
10 findByCompanyEq
11 findByCompanyEq_StatusEq
12 findByCompanyEq_AmountGt
13 findByCompanyEq_StatusEq_AtSiteEq
```

Zdrojový kód 6.1: Purchase.dd

První argument je kořen cvs stromu. Argumenty na druhé řádce identifikují jméno komponenty (*sitenet*), Javovský balík (*net.palver.uams*), jméno aktuálního datového elementu (*Purchase*) a krátké jméno aplikace (*tsms*), ve které bude element umístěn.

Řádky 3–9 specifikují jednotlivé datové položky. První částí je vždy jméno datového typu (možné jsou $Integer, Short, Long, String, Date, StringLong$ a pokud se jedná o odkaz na jiný element, tak Ln). V případě odkazu následuje jeho přesný typ:

- Ln01 – many-to-one, kde vztah obstarává softwarový NS element.
- Ln02 – many-to-one, kde se o vztah stará JEE kontejner (CMR[5]).
- Ln03 – many-to-many, kde JEE kontejner obstarává vztah (CMR).
- Ln04 – one-to-many, reverzní Ln02 vztah, který obstarává JEE kontejner (Na druhé straně vztahu musí být definován vztah Ln02.).

[5]Container Managed Relationships

- Ln05 – one-to-many, reverzní Ln01, kde obstarává vazbu NS element (Ln01 musí být specifikováno na druhé straně a jeho jméno musí být stejné se jménem Ln05. Na straně Ln01 musí být také specifikována metoda *findBy<element>Eq.*).

- Ln06 – many-to-many, kdy JEE kontejner spravuje vztah (Ln03 musí být definováno na druhé straně.).

Specifikace typu v případě odkazu je následována jménem package a třídy druhého elementu. Pokud druhý element patří do jiné komponenty, pak musí být před jménem package název archivu a speciální oddělovač, například: *<comp-name>.jar#*

Druhou částí je jméno atributu a třetí část představuje tři modifikátory, z nichž každý může být *y* (ano) nebo *n* (ne). První modifikátor označuje, jestli se atribut objeví v informativní verzi DTO objektu. Druhý označuje, jestli je pro daný atribut vyžadována vyhledávací metoda (ale tento atribut je deprecated), a třetí označuje, jestli může atribut nabývat pouze omezeného množství hodnot. Pokud ano, musí být hodnoty zahrnuty (a odděleny podtržítkem).

Řádky 10–13 specifikují vyhledávací metody, které mají být vygenerovány. Každá metoda musí začínat slovem $findBy$ a poté může obsahovat libovolné množství atributů. Tyto atributy musí být odděleny podtržítkem (ve významu AND) a jedním z následujících slov:

- Eq – rovno

- Gt – větší než

- Lt – menší než

Z tohoto popisu je jasné, jaké možnosti nabízí expander elementů, a tudíž i jaké vlastnosti od NoSQL databází požadujeme ve vztahu ke struktuře uložených dat a jejich vyhledávání. Jejich souhrn je následující:

1. Možnost uložit primitivní datové typy.

2. Možnost uložit reference na jiné objekty (vazby na další záznamy) všech typů (1:1, 1:n, m:n).

3. Vložit omezení na množinu přístupných hodnot libovolného atributu (lze zajistit i na aplikační vrstvě).

4. Možnost vyhledávat mezi záznamy podle libovolných atributů na základě jejich hodnoty (menší než, rovno, větší než).

5. Automaticky generovat jednoznačný identifikátor persistentního objektu (opět lze zajistit i na aplikační vrstvě).

V dalším popisu se zaměříme pouze na platformu Java EE. Jednak z důvodu, že je velice rozšířená, ale také proto, že ji používá i referenční řešení a můžeme z něho tedy vycházet.

Možností, jak pracovat s NoSQL databázemi z normalizovaného systému, je více. První z nich je použití stávajících expanderů (stejných jako v případě napojení na relační databázi). Tou druhou je napsání vlastních expanderů, které vygenerují odlišné zdrojové kódy (Java třídy). Oběma metodám se budeme věnovat v následujících kapitolách, nejprve si však popíšeme existující způsob propojení specifikace JPA a nerelačních databází.

Ve zbytku práce je také často odkazováno na existující expandery nebo existující implementaci NS. To nemusí vždy nutně znamenat „referenční“ řešení z Normalized System Institutu, ale jemu odpovídající ekvivalent vytvořený konkrétní společností. Předpokládá se ale, že existují expandery datových elementů do Java Entit pro použití s JPA a jejich struktura a vstupní parametry odpovídají referenčnímu řešení.

6.2 Mapování objektů do NoSQL databází

V této části popíšeme stávající možnosti mapování Java EE aplikace napsané podle standardů JPA do některých nerelačních databází. Nebudeme popisovat všechny možnosti, ale pouze několik ilustrativních způsobů, aby bylo vidět, že je to možné. Prvním velice dobrým příkladem může být Hibernate OGM (Object Grid Mapping). Hibernate je velice populární pro objektově-relační mapování, ale nabízí také zajímavé možnosti na poli NoSQL databází.

Hibernate OGM nabízí podporu JPA pro NoSQL řešení [20]. Používá Hibernate ORM engine, ale ukládá entity do NoSQL databází. Zatím jsou podporovány následující databáze:

- databáze klíč-hodnota – Infinispan, Ehcache
- dokumentové databáze – MongoDB, CouchDB (experimentálně)
- grafové databáze – Neo4j

Podle [6] se ale chystá podpora i pro Redis a další druhy databází.

Hibernate OGM podporuje několik způsobů dotazování nad daty. Jednak JPQL dotazy (automaticky se převedou do nativních databázových dotazů), potom také specifické dotazy pro danou databázi a nakonec také fulltextové dotazy. Nejlepší vlastností na tomto řešení je, že není potřeba nijak aplikaci upravovat pro použití s jinou databází než relační. Stačí pouze vzít existující kód, přidat závislost na Hibernate OGM a upravit *persistence.xml* s údaji o nové databázi. Je dokonce možné kombinovat více druhů databází najednou (tedy pro některé entity používat např. Neo4j, pro jiné Oracle nebo MongoDB). V tomto případě ale není možné používat reference mezi entitami,

které jsou spravovány různými databázemi (ale i to by mělo být v budoucnu možné [20]).

Hibernate OGM nabízí řešení i pro transakce a to dokonce i v případě, že je daná databáze nepodporuje. Je tedy možné používat standardní mechanismus transakcí, ale nemůžeme se spolehnout na rollback v případě chyby. Více informací včetně příkladů mapování do různých skupin databází lze nalézt na stránce `https://docs.jboss.org/hibernate/ogm/4.1/reference/en-US/html_single/`.

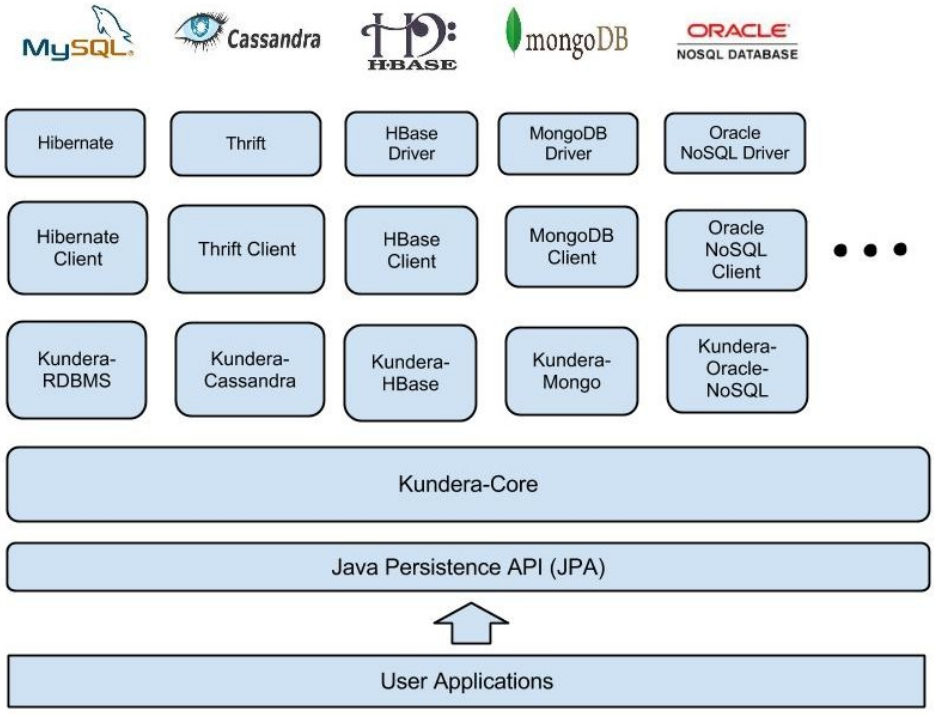
Další technologií je EclipseLink. Ta zatím podporuje mapování entit do databází MongoDB a Oracle NoSQL [3]. Celé řešení je velice podobné způsobu popsanému pro Hibernate OGM, ale vyžaduje některé nové anotace. Například každá entita, která se ukládá do NoSQL databází, musí být anotována pomocí $@NoSQL$. Jedná se tedy o rozšíření specifikace JPA, a proto není tolik univerzální.

Poslední technologií, kterou zmíníme, je projekt Kundera [13]. Kundera se striktně drží specifikace JPA 2.1 a nepřidává žádné vlastní anotace. Aktuálně podporuje následující databáze: Cassandra, HBase, MongoDB, Redis, Oracle NoSQL, Neo4j, CouchDB, Elastic Search a relační databáze. Dokáže dokonce uložit entity, které na sebe odkazují, do různých databází a pracovat s nimi napříč těmito databázemi v jednom dotazu. Díky jazyku JPQL máme možnost pracovat s různými druhy databází pomocí jednoho univerzálního jazyka založeného na SQL. Mimo tohoto jazyka podporuje také dotazování pomocí RESTu a dokonce dokáže v daných databázích generovat datové schéma. Architektura Kundery je na obrázku 6.1

NoSQL databáze mají spoustu výhod, ale také mnohé nevýhody. Mezi hlavní nevýhody patří jejich malá univerzálnost a také velké množství dotazovacích jazyků. Mnoho NoSQL databází používá Map-Reduce, ale také REST, nebo vlastní jazyky jako Cypher, FLOWER... Martin Fowler [4] uvádí termín „Polyglot Persistence“, kdy je pro každou část aplikace ideální zvolit tu nejlepší databázovou technologii odpovídající danému problému. Řešení jako Hibernate OGM nebo Kundera nám toto použití značně zjednodušují a odstiňují nás od potřeby znát mnoho různých technologií, dotazovacích jazyků a práce s nimi. Příklad, jak taková aplikace může vypadat, je na obrázku 6.2.

6.2.1 Způsoby převodu objektů do požadovaného formátu

Mapování objektů do různých modelů nemusí být vždy zcela jednoduché a někdy může vést nevhodné použití některých konstruktů ke složité nebo neoptimální datové struktuře. Následující podkapitoly ukáží, jak je možné k této transformaci přistupovat zcela obecně, nebo jak k ní přistupují technologie Hibernate OGM a další při využívání specifikace JPA. Nejedná se o úplný výčet všech způsobů, ale dostačující ukázku základních možností.



Obrázek 6.1: Znázornění architektury databázové vrstvy Kundery [13].

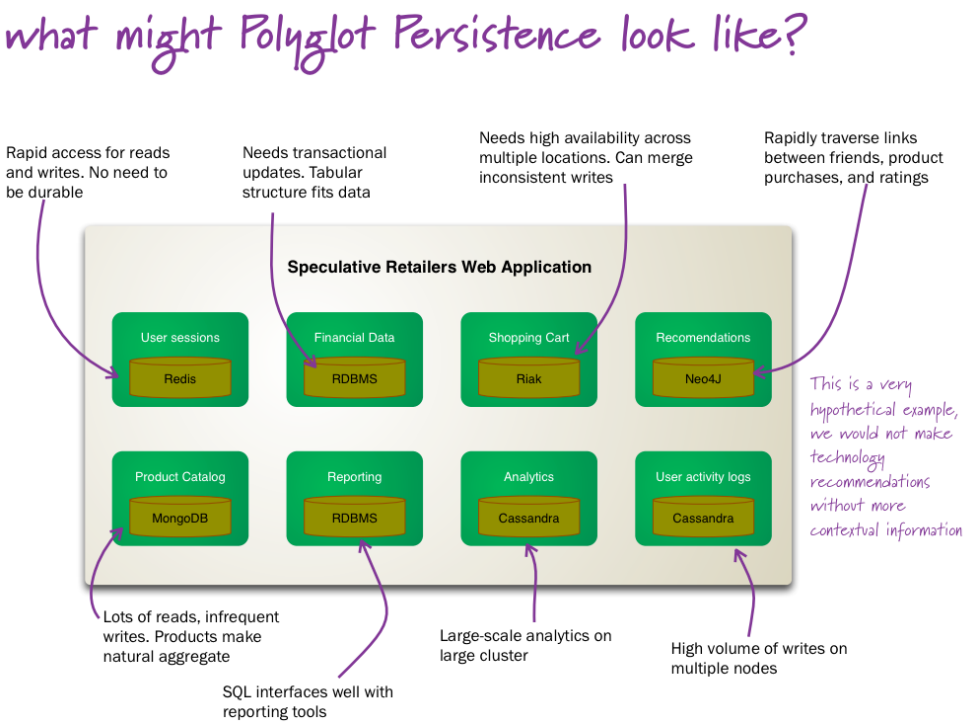
6.2.1.1 Klíč-hodnota a sloupcové databáze

Klasické databáze klíč-hodnota jsou velice jednoduché a moc se nehodí pro ukládání příliš složitých objektů s mnoha vazbami. Databáze Cassandra (sloupcová databáze) již nabízí supersloupce[6] a lze tedy mnohem lépe použít.

Každá anotace @Entity odpovídá jedné rodině sloupců (Column Family[7]) Anotace @Table(name) respektive @Column(name) přejmenovává název rodiny sloupců, respektive hodnoty v daném řádku. Položka anotovaná pomocí @Id tvoří klíč řádku. Pro vložení třídy anotované příznakem @Embeddable do atributu s @Embedded lze použít supersloupce, kde název tohoto supersloupce je název atributu a hodnota je mapa vlastností vkládaného objektu. Kolekce lze vkládat opět použitím supersloupce, ve kterém jsou jednotlivé sloupce hodnotami z kolekce. Vazby 1:1 a 1:n lze uložit jako sloupec, jehož hodnota je Id odkazovaného objektu (řádku). Vazby m:n lze uložit opět za pomoci supersloupce.

[6]Supersloupec je sloupec, který jako svoji hodnotu obsahuje další sloupce (dvojice klíč-hodnota).

[7]Column Family sdružuje řádky, které obsahují podobné sloupce.



Obrázek 6.2: Polyglot Persistence podle Fowlera [4]

6.2.1.2 Dokumentové databáze

Následující popis je zaměřen na databázi MongoDB a nemusí přesně platit pro ostatní dokumentové databáze, ale další zástupci by se neměli radikálně lišit. Ukládané dokumenty odpovídají strukturám v programovacích jazycích. Tedy přesně tomu, co v NS nazýváme datové elementy. MongoDB ukládá tyto dokumenty do kolekcí (tedy do skupin podobných dokumentů), které mohou obsahovat všechny objekty konkrétní třídy. Každý dokument obsahuje hodnoty odpovídající položkám objektů.

Anotace @Entity tedy odpovídá dokumentu v kolekci s názvem jména třídy. Anotace @Table(name) respektive @Column(name) přejmenovává název kolekce, respektive hodnoty v dokumentu. Položka anotovaná pomocí @Id je v MongoDB uložena jako hodnota _id (interní označení primárního klíče). Hibernate OGM doporučuje [20] používat pro položky Id anotaci @Type(type = "objectid") – uloží položku jako MongoDB klíč. Třída anotovaná pomocí @Embeddable se do @Embedded atributu uloží jako vnořený dokument. Kolekce lze ukládat jako vnořené kolekce (vhodné pro primitivní typy nebo kolekce objektů označených pomocí @Embedded) nebo jako kolekci _id atributů jiných dokumentů. Pořadí v kolekcích lze přidat doplněním atributu „order“ do dokumentu.

Asociace lze mapovat více způsoby. Podle [20] například:

1. uvnitř entity
2. pomocí asociativního dokumentu
3. pomocí kolekce pro každou asociaci (analogie se vztahovou SQL tabulkou)

V prvním případě je vazba uložena jako jeden odkaz nebo kolekce odkazů v odkazujícím dokumentu. Anotace @JoinColumn(name) tedy přejmenuje tuto položku. Druhou možností je použít jednu společnou asociační kolekci, do které se budou ukládat veškeré vazby. Třetí možností je použít asociační kolekci pro dokumenty ze dvou konkrétních kolekcí. U tohoto řešení lze využít anotace @JoinTable(name) a přejmenovat kolekci vazeb. Pomocí anotace @Version můžeme také označit atribut pro optimistické zamykání[8].

6.2.1.3 Grafové databáze

Opět se zaměříme pouze na jednu databázi (Neo4j), ale princip bude pro většinu grafových databází stejný. Grafová reprezentace je velice přirozená pro spoustu problémů a také se výborně hodí pro objekty, a to zejména proto, že dokáže velice dobře reprezentovat jejich vztahy. Nevýhodou pak může být vyhledávání mezi všemi objekty jedné třídy, protože jsou rozptýleny po celém grafu. Nejpřirozenější mapování je uložit objekty jako uzly a jejich vztahy jako vazby. Neo4j dokáže každému uzlu přiřadit popisky (labels), libovolný počet atributů (dvojice klíč-hodnota) a vazeb. Každá vazba je orientovaná a může opět obsahovat libovolný počet popisků a atributů.

Třída s anotací @Entity odpovídá uzlu s popiskem jména třídy. Hibernate OGM [20] označuje tyto uzly ještě popiskem :ENTITY. Třídní atributy jsou mapovány jako atributy uzlů. Anotace @Table(name) respektive @Column(name) přejmenovává název uzlu, respektive atributu v něm. @Embedded objekty mohou být uloženy jako atributy uzlu s nějakou předponou (například odděleny tečkou) nebo jako samostatné uzly s popiskem :EMBEDDED a vazbou s názvem odkazujícího atributu. Kolekce lze reprezentovat více vztahy k dalším uzlům. Pořadí v kolekci může označovat atribut vazby. Jelikož jsou asociace mapovány vždy pomocí vazby, anotace jako @JoinColumn zde nemají význam. Mapování všech druhů vazeb je velice intuitivní a není potřeba vytvářet žádné vztahové uzly nebo další pomůcky. Identifikátor s @Id může být v Neo4j unikátní constraint. Automatické generování primárních klíčů (@GeneratedValue) může obstarat uzel s uloženou hodnotou pro každý čítač.

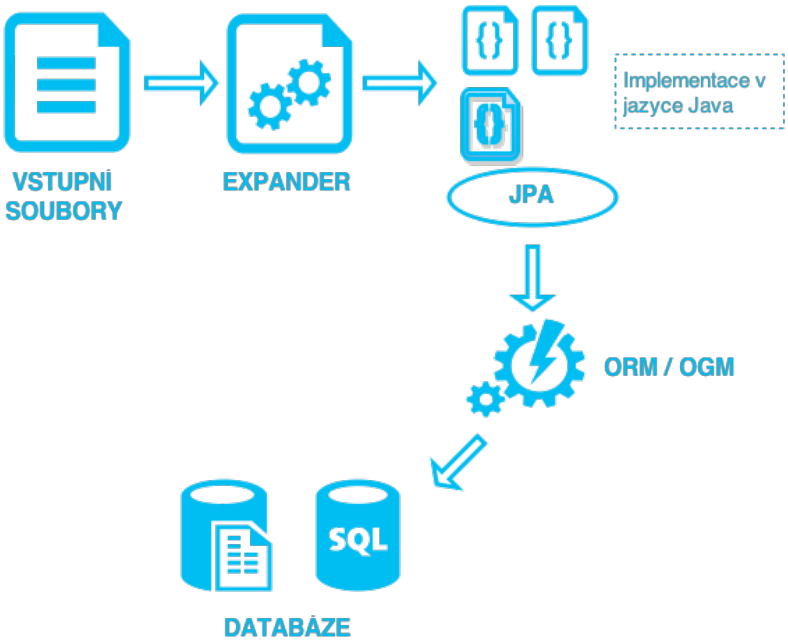
[8] „Optimistické zamykání“ nezamyká řádky při čtení, ale při každém zápisu zvýší čítač zapsání. Pokud se někdo pokusí zapsat data s čítačem nižším, než je aktuálně uložená hodnota, je vyhozena chyba.

6.2.1.4 XML databáze

Třída s anotací @Entity se může namapovat na stejnojmenný element v kořenovém elementu. Jednotlivé sloupce se mapují jako jeho děti. Tyto elementy mohou být vnořené, čehož lze využít při použití @Embedded objektů. Atribut @Id může být mapován jako samostatný tag nebo jako atribut elementu reprezentujícího objekt. Kolekce lze uložit pomocí více tagů v elementu představujícího atribut. Objekty lze odkazovat pomocí elementu obsahujícího Id odkazovaného objektu.

6.2.2 Použití existujících expanderů

Jako nejjednodušší a nejrychlejší řešení se jeví použití existujících expanderů spolu se speciální vrstvou (například Hibernate OGM), která obstará persistenci. V tomto případě nebudou potřeba žádné změny a použití databáze je zcela transparentní vzhledem k aplikaci. Stojí pouze za zvážení způsob dotazů a jejich optimalizace podle specifik jednotlivých paradigmat. Schéma datových entit je na obrázku 6.3.



Obrázek 6.3: Schéma expanze datových entit se zachováním stejných expanderů.

Je jasné, že některé NoSQL databáze (již díky svému datovému modelu) nemohou dosahovat takové komplexnosti a univerzálnosti jako klasické relační databáze. Je tedy nutné vybrat pro jejich použití pouze vhodné části aplikace a strukturu daných entit přispůsobit zvolenému modelu databáze. Ale ani

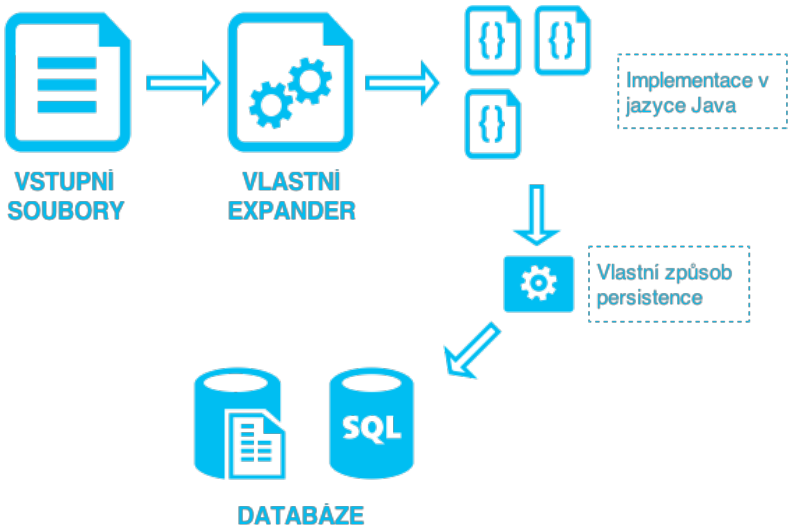
použití relačních databází není pro potřeby objektového programování vždy nejefektivnější řešení a během dlouhé praxe s nimi se vyvinulo mnoho postupů a best practices, jak je používat. Z tohoto důvodu také vznikaly ORDBMS (objektově-relační databáze) a ODBMS (objektové databáze). Výhodou Hibernate OGM je tedy možnost použít pro specifické části aplikace nejvhodnější možné databáze a vhodně z nich zkombinovat celý datový model, aniž by bylo nutné nějak výrazně upravovat kód, který se stará o persistenci.

Toto řešení může být také vhodné pro databáze, které přímo podporují standard JPA. Například objektová databáze ObjectDB [17].

Pro jednodušší aplikace, ne příliš výkonově náročné, nebo pro začátky s NS lze tedy použití OGM doporučit. Navíc je zde značná výhoda v tom, že pokud daná databáze nebude vyhovovat, lze ji vždy jednoduše nahradit jinou (i relační) a díky standardu JPA se to obejde bez kombinatorických efektů a bez nutnosti přegenerovávat třídy jinými expandery.

6.2.3 Vytvoření specifických expanderů

Druhou možností je vytvoření vlastních expanderů, které budou generovat odlišné struktury tříd databázové vrstvy a nebudou vytvářet konstrukty, které jsou požadovány standardem JPA (například anotace Entitních tříd,...). Tyto expandery mohou vycházet ze stejné struktury vstupních dat, jaká je uvedena v kapitole 6.1, nebo mohou využívat odlišný formát vstupu. Znázornění je na obrázku 6.4.



Obrázek 6.4: Schéma expanze datových entit při vytvoření nových expanderů.

6.2.3.1 Použití existujícího formátu datových deskriptorů

Jednoduchá sémantika vstupních souborů nenabízí příliš mnoho prostoru pro přizpůsobení expanderů datovému modelu NoSQL databází. Pro každou datovou položku je specifikován pouze typ a trojice příznaků. První z nich (jestli se atribut objeví v informativní verzi DTO objektu) nemá s persistencí nic společného. Druhý (jestli bude pro atribut vygenerována vyhledávací metoda) může sloužit jako pomůcka expanderu pro optimalizaci struktury tak, aby se dalo podle daného atributu lehce vyhledávat. Tento příznak je však již deprecated[9] a místo něho jsou na konci souboru definovány konkrétní vyhledávací metody. Poslední příznak specifikuje výčtový typ. Této informace lze využít při generování třídy reprezentující datovou entitu[10] a přidat všechny možné hodnoty jako databázové constrainty[11]. Většina NoSQL databází je však bez pevného schématu nebo ukládá nestrukturovaná data, a proto ani možnost zavedení constraintů neumožňuje. Toto omezení ale není problém a expander může tyto hodnoty kontrolovat již v business vrstvě při volání metody *set-*.

Jak je již zmíněno v předchozím odstavci, vyhledávací metody specifikují, podle kterých atributů a jejich kombinací se může vyhledávat. Tato informace je použitelná pro vrstvu starající se o persistenci objektů (a tudíž by měla být prostřednictvím expanderu přenesena do generované aplikace – entitní třídy), která podle ní může přizpůsobit datový model nebo vytvořit potřebné indexy.

Ničím není dáno, že konfigurace ukládání dat do databází musí být zapsána pomocí anotací nebo ekvivalentem v podobě XML konfiguračních souborů. Je možné v expanderu vygenerovat třídu, která se bude starat o ukládání konkrétní datové entity a to, které její atributy uloží a jak jejich reprezentaci převede do struktury požadované databázovým strojem, je pouze na ní. Důležité ale je, aby tato třída byla pouze jedna pro danou entitu, respektive aby vystavovala transparentní verzi. Při přidání nového persistentního atributu daného datového elementu se do odpovídající třídy, která se stará o jeho persistenci, pouze přigeneruje pravidlo pro jeho uložení. Tento způsob vede vždy pouze ke konstantnímu počtu změn, a tudíž negeneruje kombinatorické efekty.

Jak je již zmíněno výše, změna databázové technologie povede vždy ke kombinatorickým efektům (pokud se tedy nepoužije jiná technologie, ale se stejným API – jako v případě použití JPA a vrstvy Hibernate OGM), protože se jedná o změnu průřezového problému spojeného s externí technoloií, a tudíž se jí v žádném systému nelze vyhnout. V případě použití stejného formátu vstupních souborů expanderů se ale může jednat pouze o přegenerování pat-

[9]Deprecated znamená, že atribut je označen a již se nemá používat, ale zůstává zde z důvodu zpětné kompatibility.

[10]V následujícím textu představuje pojem datová entita třídu, jejíž atributy se persistují do uložiště, ale nemusí splňovat požadavky na Entity kladené technologií Java EE.

[11]Omezující podmínka pro atribut, která musí být splněna, aby bylo možné hodnotu uložit do databáze.

řičných datových elementů a výsledná náročnost změny bude pouze minimální.

Při expandování datových elementů můžeme vytvářet kód pro entity různým způsobem. Příklad srovnání dvou přístupů starajících se o persistenci datové entity pomocí nativního API databáze Neo4J a pomocí knihovny Spring Data Neo4J (SDN) [25] může vypadat tak, jak je znázorněno v kódu 6.2 a 6.3 [24]:

```
1  public class User {
2
3    private final Node underlyingNode;
4
5    public User(final Node node) {
6      underlyingNode = node;
7    }
8    public Node getUnderlyingNode() {
9      return underlyingNode;
10   }
11   public final String getName() {
12     return (String) underlyingNode.getProperty("name"
         ↪ );
13   }
14   public void setName(final String name) {
15     underlyingNode.setProperty("name", name);
16   }
17 }
```

Zdrojový kód 6.2: Bez použití SDN

```
1  @NodeEntity
2  public class User {
3
4    @GraphId
5    private Long id;
6
7    @Indexed(unique = true)
8    private String name;
9
10   @RelatedTo(type="FRIENDS", direction = INCOMING)
11   Set<User> friends;
12
13   // getters and setters
14 }
```

Zdrojový kód 6.3: S použitím SDN

6.2.3.2 Použití specifického formátu datových deskriptorů

Poslední možností je nejen přepsat vlastní expandery, ale také změnit jejich vstupní soubory. Tato varianta je rozhodně nejkomplexnější a nejflexibilnější.

Výhodou tohoto řešení je možnost přizpůsobit konfigurační soubory specifické databázové technologii a optimalizovat tak celý proces persistence (ať už z hlediska rychlosti, velikosti nebo výpočetní náročnosti). Rychlost IS však není hlavní devizou NS a ustupuje jejich rozšiřitelnosti a schopnosti reagovat na změny.

Nevýhodou je nemožnost změny databázové technologie, která by používala jiný model pro ukládání dat (např. změna z databáze klíč-hodnota na grafovou databázi) bez vzniku kombinatorických efektů. Respektive změna bez těchto efektů by možná byla, ale bylo by nutné naimplementovat expander ze specifického formátu vstupních souborů do jiné technologie. Předpokládáme-li však vznik speciálního formátu popisu datových elementů z důvodu vyšší efektivity při jejich ukládání, není pravděpodobné, že by tento formát vyhovoval zcela jinému modelu dat. Spíše lze očekávat, že pro dosažení požadované efektivity bude nutné tento formát opět zcela změnit.

Někdo by mohl argumentovat tím, že potřeba změny databázové technologie v dané aplikaci nikdy nenastane, a tudíž nevznikne existence kombinatorických efektů. Je zde však stále možnost, že používaná databáze změní svoje API nebo způsob ukládání dat a expandery včetně jejich vstupních souborů se stejně budou muset upravit. Druhým a možná zcela nejdůležitějším argumentem proti použití této varianty napojení aplikace na NoSQL databáze je to, že popis datových elementů by měl být zcela nezávislý na použité technologii. Vstupní soubory by tedy neměly obsahovat popisy nebo rady, jak jednotlivá data ukládat, ale měly by sloužit čistě jako popis dat (jejich obecná struktura). To, jak s nimi aplikace naloží, je zcela na ní.

Existuje zde ještě varianta, kdy se ponechá formát vstupních souborů tak, jak je popsán v sekci 6.1 a s kterým dokáží pracovat existující expandery, a pouze se rozšíří o nové položky, které neovlivní expandery, které s nimi nepracují. Toto řešení můžeme brát jako kompromis mezi efektivitou a udržovatelností.

6.3 Požadavky na transakční zpracování

Samotná teorie NS žádné požadavky na transakční zpracování neklade a v žádném pravidle není řečeno nic o nutnosti použití transakcí. Nicméně referenční implementace je díky technologii JPA používá, a proto je možné, že se jejich použití implicitně předpokládá.

Použití transakcí není záležitostí konkrétní teorie, ale jde spíše o vlastnosti kladené na systém ze strany funkčních a nefunkčních požadavků. Je jasné, že při paralelním přístupu k datům může docházet k současnému přístupu ke stejné položce, a pokud se tato položka na obou místech edituje,

vznikne konflikt, který může vést k některým anomáliím popsaným v kapitole 2.1. Těmto anomáliím lze teoreticky předejít linearizací prováděných operací, ale tento přístup naprosto degraduje dostupnost a u dlouhotrvajících business transakcí je tento přístup naprosto vyloučen. Je tedy jasné, že se konflikty vyskytují a musí se nějak řešit. Někdy je nutné zaručit atomicitu každé transakce a jindy naopak můžeme povolit ne zcela konzistentní stav databáze. Jak je již zobrazeno na obrázku 6.2 v kapitole 6.2, pro každou specifickou část aplikace se hodí jiná databáze s odlišným přístupem k transakcím, dostupnosti, distribuovanosti,... U každé části (modulu, služby,...) systému musíme tedy rozhodnout, kterou vlastnost z CAP teorému budeme preferovat. Pokud se rozhodneme pro dostupnost a ohleduplnost k rozdělení, může se nám stát, že některý záznam nebude zcela validní nebo nepůjde vůbec uložit. S tím se ale musíme vypořádat v samotné aplikaci nebo se spokojit s malým procentem chybných položek v uložišti. Řešení problému plynoucího z CAP teorému není nemožné, ale není ani jednoduché a nelze generalizovat. Vždy záleží na konkrétní aplikaci a stavu, ve kterém se nachází. Některé postupy, jak se vypořádat s nemožností splnit všechny tři požadavky CAP teorému, jsou uvedeny v kapitole 2.3. Základem jsou ale krátké transakce.

Při použití krátkých transakcí se snižuje pravděpodobnost jejich konfliktu a tím i vzniku anomálií. Této potřebě značně nahrává teorie NS. Díky větě 4 musí každá akční entita udržovat stav zpracování daného datového elementu. Je tedy nutné po každé provedené operaci uložit její výsledek. Jak je již zmíněno výše, toto uložení může být i persistentní. Jedná se tedy o rozdělení zpracování na mnoho malých částí s průběžným ukládáním výsledku. Uvážíme-li ukládání těchto výsledků persistentně, máme log všech operací a jejich výsledků, které proběhly v aplikaci. Uložení chybového výsledku zpracování nám tedy umožní nechat nějakou akční entitu na tento výsledek reagovat a tím se můžeme vypořádat i s transakčními anomáliemi. Entita starající se o zpracování konkrétního chybového stavu může provést rollback odpovídající akce na úrovní aplikační logiky nebo se pokusit provést akci znovu. Tímto způsobem lze také obnovit konzistenci nebo opravit porušený stav databáze. Nejedná se tedy o univerzální řešení chybového stavu pomocí akce rollback, ale tento způsob nám dovolí na odlišné chybové stavy reagovat odlišným způsobem. To odpovídá pohybu po stavovém automatu, ve kterém jednotlivé stavy představují stav aplikace, což značně redukuje komplexnost řešení, protože se můžeme zabývat pouze malým okolím daného stavu.

Jak je tedy vidět, NS jako takové nejen nekladou žádné požadavky na transakční zpracování, ale dokonce vedou k jejich omezení. Pokud navíc připočteme možnost pracovat s „transakcemi“ na aplikační vrstvě pomocí technologií jako Hibernate OGM nebo Kundera, nemusíme se o problém konzistence příliš starat a o jejich potřebě rozhodnout pouze na základě business požadavků.

6.4 Spolupráce s jinou platformou než Java EE

Jak již bylo uvedeno v kapitole 6.1, zabývali jsme se pouze technologií Java EE. To ale není podmínka a stejného výsledku lze dosáhnout v libovolném jazyku. Ať už se jedná o jiný objektově-orientovaný jazyk, tak i jazyk s odlišným paradigmatem. Většina jazyků provozovaných nad JVM (Java virtual machine) umí spolupracovat s Javou, a tedy i využívat technologie jako JPA nebo Hibernate. Další jazyky obvykle také nabízí ORM a stejným způsobem lze přidat podporu i pro ostatní druhy databází. Pro použití mapování založeného na struktuře dat není potřeba tříd, ale stačí pouze datové struktury, které obsahují pouze datové položky. ORM, OGM a další principy zde mohou fungovat naprosto stejným způsobem. Pro objektové mapování není potřeba ani speciálních anotací. Místo nich lze stejně dobře použít XML nebo jiné konfigurační soubory nebo mapovací funkce. Při přidání nové položky do struktury se pouze přidá další pravidlo (náhrada anotace) do konfiguračního souboru a počet změn zůstává konstantní.

Není ale nutné používat pouze technologie na způsobu anotování položek, které se mají persistovat do uložiště. Další zcela legitimní možností je napsání funkcí, které budou persistenci řešit explicitně. Důležité ale je, aby se mechanismus ukládání konkrétní položky vyskytoval pouze na jednom místě v celém programu. Lze tedy napsat metodu, ve které bude sestavován dotaz (nejlépe s využitím builderu dotazu) nad daným uložištěm a ve specifickém jazyce (například pro uložení nového uzlu pomocí jazyka Cypher). Důležité ale je, aby tento dotaz existoval pouze jeden pro uložení jedné entity. Kdyby jich totiž existovalo více, přidání nové persistentní položky entity by vedlo k n změnám ve všech metodách starajících se o persistenci této entity. Jediný rozdíl je u vyhledávacích metod. Těch může být více, ale jsou specifikovány jako vstup při expandování datových elementů a mohou být později přidány nové.

V zásadě není nutné mapovat třídy do databáze přesně tak, jak je tomu v případě entit. Při použití v NS je pouze nutné, aby objekt, který vznikne vytažením dat z databáze byl zabalený datový element a aby měl transparentní verzi. Třída starající se o persistenci libovolného objektu může tento objekt před uložením rozložit na více částí (tabulek, dokumentů, uzlů, dvojic, ...) a při vyhledávání ho opět poskládat nazpět. Je ale nutné, aby se tato transformace vyskytovala pouze na jednom místě. Obecně by se tedy dalo říci, že je potřeba vytvořit mechanismus, který dostane datovou entitu a dokáže ji persistovat, nebo naopak dostane nějaké parametry a na jejich základě dokáže tuto datovou entitu vrátit.

Kapitola 7

Rozbor možností nabízených jednotlivými databázovými platformami

Jak již plyne z předchozích kapitol, zejména z kapitoly 2.4, NoSQL databáze se od sebe hodně odlišují a výběr té ideální pro konkrétní nasazení nemusí být vždy lehký. Zejména také kvůli omezené komplexnosti a robustnosti těchto databází a jejich datových modelů se nehodí pro tak široký záběr nasazení jako relační databáze a jejich nevhodné použití může zničit nejen konzistenci, ale také výkonnost (o kterou nám u NoSQL databází jde především). Příkladem některých nevhodných operací může být vyhledávání v nestrukturovaných datech, rozsáhlé spojování silně provázaných dat nebo přístup k zanořeným položkám jednotlivých záznamů. Dalo by se tedy říci, že je mnohem závažnější rozhodnout se mezi použitím databáze Cassandra nebo Neo4j než mezi PostgreSQL a Oracle.

7.1 Klíč-hodnota databáze

Tyto databáze se hodí pro jednoduché objekty bez velkého zanořování a s malým množstvím referencí. Často se používají pouze pro caschování dat. Jejich použití pro datový model celého systému je nevhodné. Naopak se výborně hodí tam, kde bude generováno velké množství dat, jako například logování různých akcí provedených v rámci systému (notifikace, odeslané e-maily, historie procházení,...). Často se také může jednat pouze o vestavěné databáze, které neběží ve vlastním procesu, ale jde pouze o knihovnu, která sdílí proces se samotnou aplikací. Tím se snižují nároky na výkon a administraci.

V teorii NS lze tyto databáze dobře použít například pro uchovávání stavu workflow elementů a ukládání výsledků po dokončení akčních elementů. Zde se předpokládá mnoho zápisů (aplikace bude po každé provedené funkci něco

zapisovat) a velmi malá provázanost hodnot. Každý výsledek je vždy spojen pouze s jedním datovýmn elementem. Zde nebude vadit ani často se vyskytující neschopnost dodržet konzistenci, protože se data pouze zapisují a čtou (neprobíhá žádná modifikace).

7.2 Dokumentové databáze

Dokumentové databáze se hodí tam, kde se přímo v aplikaci pracuje s dokumenty. Například při odesílání dat pomocí JSON souborů (RESTové služby). Dobrým příkladem může být automatické ukládání nevalidních formulářů a jejich verzování. Jejich struktura umožňuje snadné uložení složitých objektů včetně mnoha přidružených údajů. Díky vnoření dat uvnitř dokumentů se sice dopustíme denormalizace, ale za cenu vyššího výkonu a lokalizace dat. Díky tomuto přístupu lze mnohem lépe provádět distribuce dat na základě reálné geografické polohy uživatelů systému. Není tedy, například pro vyčtení adresy konkrétního uživatele, potřeba přistupovat do nějaké globální tabulky adres (společné pro všechny uživatele), ale každá osoba může mít svoji adresu přímo ve svém dokumentu a tyto dokumenty mohou tedy být umístěny na geograficky bližších serverech.

Z teorie NS nevyplývá místo, kam by se přímo hodily tyto databáze, ale jejich potřeba nebo vhodné použití může vyplynout z požadavků na samotnou aplikaci a strukturu dat v ní. Důležité ale je, že NS nekladou žádná omezení, kvůli kterým by nebylo možné tuto kategorii databází použít.

7.3 Grafové databáze

Grafové databáze se dobře hodí pro modelování vztahů. Například mezi uživateli lze jednoduše vyhledávat vzájemné přátele, jejich podobné činnosti nebo zájmy. Obecně by se dalo říci, že grafové databáze se hodí pro malé objekty s mnoha vazbami. Tyto databáze také často obsahují funkce známé z teorie grafů, jako je například prohledávání stavového prostoru (do hloubky nebo do šířky), nalezení nejkratší cesty a další. Lze tak velice jednoduše vyřešit problémy, které by se za pomoci klasického SQL řešily velmi obtížně pomocí rekurzivních dotazů nebo by musely být přeneseny do aplikační vrstvy. Naopak ne příliš vhodné jsou tyto databáze pro složité dotazy na základě mnoha hodnot jednotlivých atributů a složité agregační a statistické funkce.

Opět se v teorii NS nenachází část, pro kterou by bylo na první pohled vhodné použít právě tento typ databází, ale podobně jako u dokumentových databází si tuto potřebu mohou vyžádat vlastnosti a struktura vytvářené aplikace. Jelikož teorie grafů je velmi komplexní a široce používaná vědní disciplína, jsou databázové stroje postavené na této teorii poměrně komplexní a flexibilní. Bez větších problémů lze na těchto platformách dosáhnout stejných nebo velice podobných vlastností, které nám nabízí klasické relační databáze.

Jejich použití je proto vhodné pouze za účelem zjednodušení daného problému (někdy reprezentace grafu odpovídá reálnému vztahu elementů) nebo z výkonnostního hlediska daného strukturou dat (prohledávání sousedů,...). Rozhodně lze ale říci, že funkčně ekvivalentního řešení lze dosáhnout použitím jak relačních, tak grafových databází a konečná volba může být vykonána na základě benchmarku[12] daných databází v konkrétním nasazení.

7.4 Databáze kategorie CA

U těchto databází vítězí konzistence a dostupnost nad tolerancí k rozdělení. Jelikož do této kategorie spadají téměř všechny relační databáze (všechny obecně známé), je jasné, že se pro NS dají použít (stávající implementace je používá). Jak již bylo zmíněno, teorie NS nemá žádné požadavky, které by nedovolovaly použít libovolný datový model databází, lze říci, že je možné použít všechny databáze z této kategorie. Ne vždy však musí být omezení kladené datovým modelem adekvátní vzhledem k výkonnosti, ale volbu konkrétní technologie neomezí splnění vlastností CA, ale výše diskutované aspekty NoSQL databází.

7.5 Databáze kategorie CP

Již podle Brewera [2] je distribuovanost velice vzácná a často není potřeba. Pokud je ale aplikace dostatečné velká a vyžádá si použití více databázových serverů, může se objevit i požadavek na toleranci k rozdělení. Jak je popsáno v článku reflektujícím CAP teorém po dvanácti letech [2], můžeme se s přerušením komunikace mezi několika uzly nebo skupinami uzlů vypořádat více způsoby. Kategorie CP ale zaručuje to, že data budou synchronizována na všech uzlech i za cenu toho, že budou po nějakou dobu v celé nebo pouze v části aplikace nedostupné. Tento přístup může sice způsobit problémy s výkonem databáze (potažmo aplikace), což může být nepřípustné z pohledu nefunkčních požadavků, rozhodně ale nemůže působit problémy ve vztahu s teorií NS, protože data jsou stále konzistentní (jako u relačních, potažmo CA databází).

Dostupnost dat není něco, co by teorie přímo obsahovala, ze způsobu provádění akcí v NS se ale zdá, že tato vlastnost nebude zcela stěžejní. Tato domněnka plyne z toho, že je vhodné a doporučované [15] používat asynchronní provádění akcí. Pokud se tedy v nějakém workflow provede daná akce, je její výsledek uložen a na základě tohoto výsledku jsou spuštěny trigger elementy, které vykonávají další akce. Bude-li tedy databáze dočasně nedostupná, znamená to, že akce nezávislé na výsledku vzniklém během rozdělení mohou běžet dál a ty ostatní se vykonají, až bude výsledek dostupný. Z toho plyne, že použitím asynchronního způsobu provádění akcí, ve kterém se obecně předpokládá,

[12] Benchmark je metoda srovnání více jednotek (programů, algoritmů,...) na základě výsledků při jejich měření pomocí testů.

že výsledek bude dostupný někdy v budoucnu, se vyhneme problému s nedostupností dat. Nedostupnost dat nám totiž pouze pozdrží výsledek (sníží responzivitu), ale neovlivní běh aplikace po funkční stránce (nedoje k timeoutu atd.).

7.6 Databáze kategorie AP

Dalo by se říci, že databáze kategorie AP jsou zaměřeny na výkon. Je to dáno zejména vlastností dostupnosti. Také použití distribuovaných databází značí potřebu maximálního výkonu. V tomto případě není zcela jasné, jestli je nejvhodnější použít normalizované systémy, protože jejich schopnost adaptovat se na požadované změny bude vždy částečně na úkor výkonu. Nicméně samotný vliv NS na výkonnost by neměl být extrémně velký a testování potřebné režie k zajištění normalizace není náplní této práce. Odhlédněme tedy od vhodnosti či nevhodnosti použití NS pro potřeby vysoce dostupných systémů a předpokládejme, že i v této kategorii mají NS své místo. Oželení konzistence na úkor dostupnosti je pro celou řadu systémů ne zcela vhodné a vyžaduje odlišný a zatím ne příliš rozšířený pohled na databáze.

Přístup AP opět nelze zcela generalizovat a není možné říci, zda se pro NS hodí, nebo ne. Jsou situace, kdy uvolnění konzistence nemusí vadit, ale naopak je požadována rychlá odezva. Jde zejména o takzavaná BigData[13]. Pokud tedy přijmeme omezení plynoucí z chybějící podpory konzistence, nekladou NS žádnou překážku pro použití databází z této kategorie. Dokonce by se dalo říci, že některé nekonzistence ani nemohou vzniknout díky zjednodušeným požadavkům na data NS. Například požadavek na unikátnost daného atributu není možné do expanderu zanést a tudíž se nepropíše ani do databázové vrstvy aplikace, a není tedy možné, aby databáze porušila konzistenci tím, že bude obsahovat dvě položky se stejnou hodnotou. A těchto příkladů (zvláště u constraintů) by se dalo nalézt více.

7.7 Výběr vhodné databáze

Výběr databáze pro konkrétní část aplikace je často velice obtížný a obvykle ani neexistuje volba, pro kterou by platilo, že je nejvhodnější pro všechny požadavky najednou. Mluvíme-li v kontextu normalizovaných systémů, bude naší prioritou nejspíše přehlednost a rozšiřitelnost. Zejména přehlednost není dost často na příliš vysoké úrovni u relačních databází. Díky tomu, že se při vytváření databázových schémat často snažíme dodržovat alespoň 3NF[14], obsahují

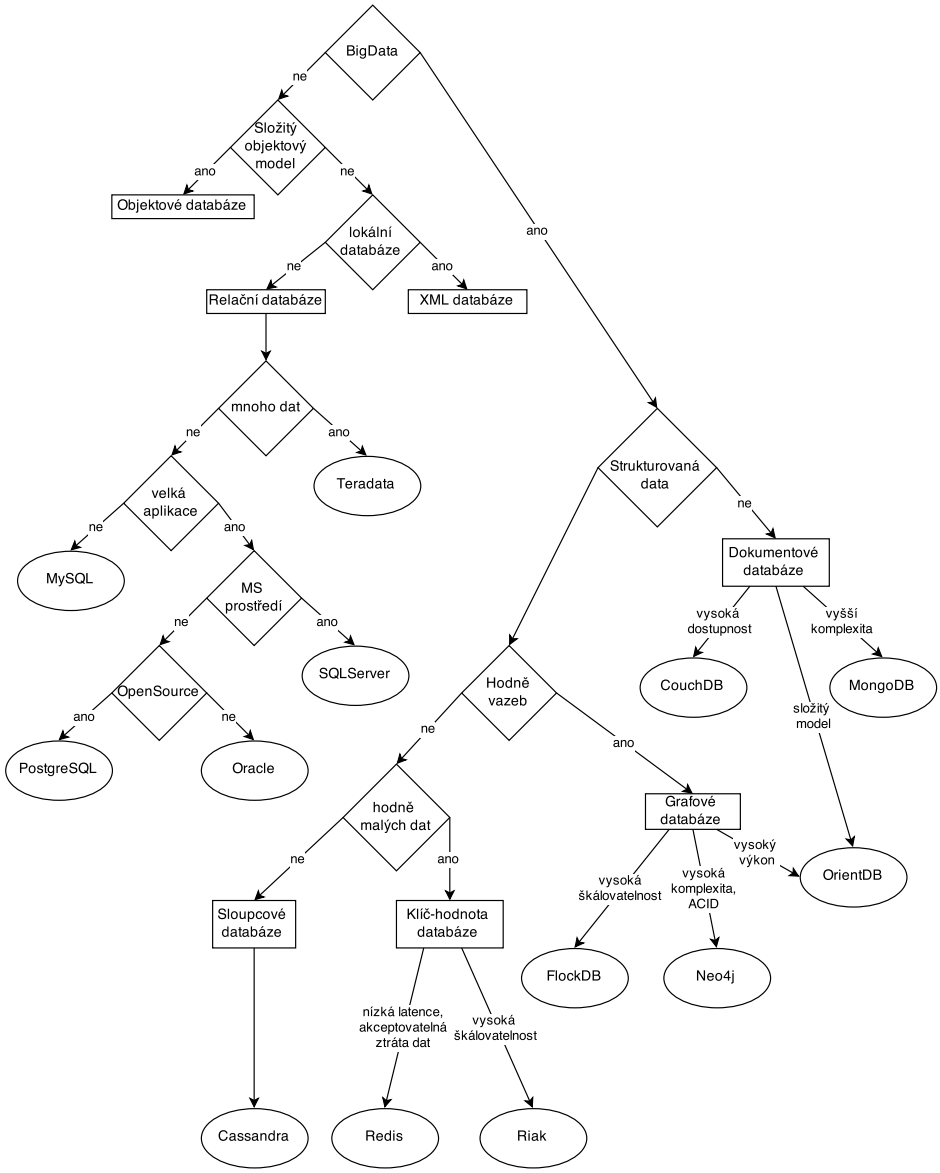
[13] Big data je označení pro rychle přibývající data nebo pro data velkého objemu. Tento pojem má více definic a žádná není zcela přesná.

[14] 3NF (třetí normální forma) – popisuje požadavky, které musí splnit každá tabulka, aby mohlo být schéma označeno jako normalizované. Existují i další normální formy (1NF, 2NF, BCNF).

tabulky mnoho nicneříkajících cizích klíčů, které snižují porozumění uloženým datům. Další nevýhodou je pevné schéma tabulek, kvůli kterému jednotlivé řádky často obsahují mnoho prázdných sloupců (nebo sloupců s hodnotou *null*). Z tohoto pohledu jsou mnohem vhodnější grafové nebo dokumentové databáze, nebo databáze bez pevného schématu.

Pokud tedy používáme pro mapování do libovolné databáze společné rozhraní (například JPA), můžeme rozhodnutí udělat pouze na základě struktury dat, která budeme ukládat, protože o vytvoření „schématu“ se postará mapovací vrstva. Budeme-li ale pracovat s databází pomocí jejího nativního API, musíme její schéma navrhnout sami a zde může špatný návrh vést k mnoha kombinatorickým efektům v budoucnosti. Aby se tomuto chování dalo zabránit, musíme strukturu dat normalizovat. Je potřeba zmínit, že se zde slovo „normalizace“ vyskytuje v jiném kontextu než je použito u NS. U NS se normalizace bere jako způsob vytváření struktury dat tak, abychom se změnou této struktury vyhnuli kombinatorickým efektům. Kdežto při normalizaci databází se nesnažíme vyhnout úpravám při změně struktury dat, ale snažíme se zamezit potřebě editace dat (záznamů) na více místech při úpravě konkrétních položek. Nejedná se tedy o změnu struktury, ale změnu obsahu dat.

Příklad možného postupu při výběru databázové technologie je zobrazen na obrázku 7.1. Je však jasné, že na základě odpovědi na jednu otázku není možné vybrat vždy ideální technologii, ale tento graf může posloužit jako pomůcka při rozhodování. Tento rozhodovací strom by se dal rozšířit i o další aspekty, ale ani tak by rozhodnutí nemuselo být vždy přesné. Musíme brát ohled i na použitou technologii aplikace, licenci databáze, OS serverů a mnoho dalších faktorů. Nakonec i pojmy jako „BigData“,„mnoho dat“ nebo „hodně vazeb“ jsou velice vágní a jejich hodnocení je vždy relativní.



Obrázek 7.1: Postup při výběru vhodné databáze.

Kapitola 8

Shrnutí požadavků

V této kapitole shrneme všechny požadavky na data v normalizovaných systémech a na příkladech ukážeme některé z možných způsobů použití. Nejprve je nutné si uvědomit, co vyjadřuje definice kombinatorických efektů. Rozlišení změn, jejichž dopad je závislý pouze na charakteru změny, je velice obtížné. Jak přesně odhadnout, jaký je charakter a velikost změny? Jedná se pouze o změny, jejichž dopad je asymptoticky odlišný? Nebo i konstantní počet změn může vést ke kombinatorickým efektům? Pokud vždy, když změníme element, budeme muset promítnout tuto změnu do deseti kusů kódu, jedná se o kombinatorický efekt? A co když těchto míst bude vždy 100? Nebo když se rozhodneme změnit nějaký průřezový problém? Asi nejlepší představa je dívat se na kombinatorické efekty jako na asymptoticky odlišné dopady. Pokud například chceme přidat jeden atribut do třídy, měl by počet změn být konstantní. Pokud ale chceme změnit přístupová práva pro n akcí, měla by složitost být lineární vzhledem k n. Velikost systému tedy nesmí ovlivnit dopad změny.

Od dat tedy potřebujeme naprostou transparentnost vůči přidání nového elementu (můžeme si představit třídu v OOP) nebo nového atributu (položku třídy). Pokud tuto položku budeme používat (číst, editovat, nastavovat) na n místech, musíme přidat patřičný kód na všech n míst. Pokud si ale přidání atributu vyžádá editaci i někde, kde se tento nový element nepoužívá, jedná se o kombinatorický efekt. Tento požadavek může například způsobit přidání tohoto atributu do konstruktoru (nebo jiné metody). V tomto případě je nutné opravit kód na všech místech, kde se používá daná metoda, aniž by se tam nový atribut nastavoval.

Teorie NS a její pokročilý model informačního systému požaduje od dat možnost uložit skupinu položek. Tato skupina může sestávat z primitivních datových typů a referencí (odkazů) na jiné elementy. Primitivní datové typy nejsou v teorii nikde přímo specifikovány a proto nejsou ani omezující podmínkou pro výběr databáze. Předpokládá se však rozumné minimum typů (celé číslo, řetězec, reálné číslo, logická hodnota). Existující řešení počítá s typy

Integer, Short, Long, String, Date, StringLong. Není však vždy nutné rozlišovat například Integer a Short nebo je možné použít Date jako referenční typ. Zde se jedná pouze o optimalizaci (velikosti nebo rychlosti), ale pokud daná databáze některý typ nepodporuje, nejedná se o větší problém a lze ho nahradit jeho nadtypem (Short → Integer → Long, String → LongString), nebo referencí (například Date). Reference mohou být přímo podporované databází (například v grafových nebo objektových databázích) nebo pomocí cizího klíče a Id záznamu (relační, dokumentové, klíč-hodnota databáze). Výsledný efekt je stejný a pro vytvoření struktur nebo objektů v aplikaci se musí postarat databázová vrstva.

Pro načítání referencovaných objektů lze použít eager[15] (včasné) i lazy[16] (líné, nebo také odložené) načítání. Zde záleží, jestli bude odkazovaný objekt používán (je tedy lepší, aby byl načten ihned), nebo ne. Toto rozhodnutí může záviset na aplikační vrstvě, ale musíme dát pozor, aby jeho změna nezpůsobila kombinatorický efekt, kdy by bylo potřeba načíst dané objekty na více místech. Obecně (i z výkonnostního hlediska) lze doporučit lazy načítání, pouze například u vazeb 1:1, kdy jsou si objekty silně svázány (např. uživatel a jeho adresa) nebo u kolekcí primitivních typů (např. seznam klíčových slov) by mohlo být vhodnější eager načítání. Je ale potřeba brát v úvahu, že pokud změníme načítání atributu z eager na lazy, bude nutné na všech místech, kde se pracuje s hodnotou daného atributu, načíst jeho hodnotu a to je kombinatorický efekt. Tento efekt ale odstraní databázová vrstva, pokud při přístupu k hodnotě lazy atributu jeho obsah transparentně vůči aplikaci načte. To je případ klasického ORM.

Skupina hodnot je tedy datová entita a mělo by být možné ji persistovat. Aby byly zaručeny všechny předpoklady NS, je datová entita (persistentní objekt, Java Entita) „obalena“ datovým elementem. Tento element by měl obsahovat persistentní jednotku, funkce starající se o persistenci, transportní objekty (detailní, se všemi položkami a informativní) a funkce starající se o převod do těchto transportních objektů. Samotný persistentní objekt musí mít označeny atributy, které se mají persistovat. Toto označení může být v podobě anotace, konfiguračního souboru, výčtu hodnot nebo jiným způsobem. Databázová vrstva tedy ví, které atributy persistovat a které ne (například EJB persistuje všechny atributy kromě těch, které jsou označeny anotací @Transient).

Transportní objekty (detailní a informační) jsou používány aplikací. Aplikace tedy nevidí samotnou persistentní entitu, ale pouze tyto dva objekty. Díky tomu je možné změnit entitu bez zásahu do zbytku aplikace. Detailní DTO obsahuje všechny atributy a informační DTO pouze ty, které jsou ve vstupních souborech označeny jako Info. O jejich vytvoření se starají samostatné třídy

[15] Eager vazba znamená, že odkazovaný objekt se načte ihned při načítání daného objektu.
[16] Při lazy vazbě se odkazovaný objekt načte, až když je k němu přistupováno.

(projektory). Tyto projektory musí obsahovat metody pro vytvoření entity z transportního objektu a naopak. Transportní objekty jsou užitečné například při přidání nového povinného atributu do persistentní třídy. V takovémto případě nemusí být tento atribut nastaven všude, kde je entita vytvářena, ale může se například dopočítat při konstrukci entity z transportního objektu v projektoru. Informační verze DTO může sloužit všude tam, kde nejsou potřeba všechny detaily objektu a může být čtena mnohem častěji. Toho lze využít například při cachování. Transportní objekty jsou také potřeba z důvodu bezpečnosti. Není například možné ve webové aplikaci posílat ze serveru na klienta celou entitu uživatele i s heslem.

Třída, která se stará o persistenci, používá databázovou vrstu. Tato třída (*-Cruds*) obsahuje metody pokrývající CRUDS operace. Každá metoda bere jako svůj parametr transportní objekt, pomocí projektoru z něho vytvoří persistentní entitu a pomocí databázové vrstvy (v prostředí JPA se jedná o třídu EntityManager) ji uloží do databáze. V případě chyby zapíše chybový stav, na který mohou ostatní jednotky reagovat.

Vyhledávání objektů v uložišti je ve stávající implementaci řešeno separátní třídou pro každou *findBy-* metodu ze vstupního souboru. Třídy obsahují atributy, podle kterých se vyhledává. Objekty těchto tříd jsou předány vyhledávací metodě v *-Cruds* třídě a ta se postará o nalezení odpovídajících položek a vrácení kolekce transportních objektů. Ve stávající implementaci je z této *-Cruds* třídy volána metoda *find* na *-Finder* třídě a na základě parametru (objektu jedné z *-findBy* tříd) se zavolá další metoda, která za pomoci třídy *QueryBuilder* vytvoří dotaz a dotáhne data. Zde je tedy možné naimplementovat vlastní *-QueryBuilder* pro potřeby NoSQL a metody, které nastavují jeho parametry, nemusí vytvářet syntaxi SQL, ale libovolného jiného jazyka.

8.1 Příklad datového elementu

V příkladu se přidržíme principů platformy Java EE a práce s databází bude pomocí mapování objektů na základě anotací. Všechny příklady budou psány pouze pseudokódem a povětšinou pouze hlavičky metod.

Příklad bude ukázán na dvou entitách *Car* a *CarModel*. Způsob řešení vychází z existujícího řešení pro relační databáze [19] a celá struktura je zjednodušena pro lepší pochopení. Vstupní soubory by mohly vypadat tak, jak je znázorněno v kódu 8.1 a 8.2.

Máme tedy dva datové deskriptory, každý definuje několik atributů a vyhledávacích metod. Zjednodušená struktura datových elementů (bez rozhraní) by mohla vypadat následovně:

Základem jsou samotné entity *CarData* (kód 8.3) a *CarModelData* (kód 8.4). Tyto objekty jsou ukládány do databáze pomocí EJB třídy *EntityManager*. Pro potřeby NoSQL databází může být tato třída nahrazena vlastní

```
eurent net.janeclu1.eurent Car
Ln02net.janeclu1.eurent.CarBrand brand ynn
Ln02net.janeclu1.eurent.CarModel model yyn
String licensePlate yyn
findByModelEq
findByLicensePlateEq
```

Zdrojový kód 8.1: Car.dd

```
eurent net.janeclu1.eurent CarModel
String name ynn
Ln02net.janeclu1.eurent.CarBrand carBrand yyn
Short dayRentalRate yyn
findByCarBrandEq
findByDayRentalRateLt
findByDayRentalRateGt
```

Zdrojový kód 8.2: CarModel.dd

implementací, která místo SQL používá specifický dotazovací jazyk nebo API dané NoSQL databáze.

```
@Entity
class CarData {

  CarData(){}

  @Id
  Long id;
  @ManyToOne
  CarBrandData brand;
  @ManyToOne
  CarModelData model;
  String licensePalete;

  // getters and setters
}
```

Zdrojový kód 8.3: CarData

Jak bylo popsáno výše, tyto třídy (*CarData* a *CarModelData*) mohou mít odlišné anotace zcela podle potřeb konkrétní NoSQL databáze. Není tedy problém místo anotace *@Entity* napsat třeba *@Document* nebo přidat další (například jméno kolekce, atributu, klíče, ...).

Dalšími třídami jsou transportní objekty. Zde se nejedná o nic složitého, a proto stačí princip ukázat pouze na entitě *Car* (kód 8.5). Oba objekty (detailní

```
@Entity
class CarModelData {

  CarModelData(){}

  @Id
  Long id;
  String name;
  @ManyToOne
  CarBrandData carBrand;
  Short dayRentalRate;

  // getters and setters
}
```

Zdrojový kód 8.4: CarModelData

i informativní) by byly v tomto případě stejné, a proto je ukázán pouze příklad *CarInfo*. Třída *DataRef* reprezentuje odkaz na další entitu a obsahuje atributy specifikující balík, entitu a její Id.

```
class CarInfo {

  Long id;
  DataRef brand;
  DataRef model;
  String licensePalete;

  // getters and setters
}
```

Zdrojový kód 8.5: CarInfo

Pro sestavování transportních objektů slouží třídy *CarDetailsProjector*, *CarInfoProjector* a ekvivalenty pro *CarModel* (*CarModelDetailsProjector* a *CarModelInfoProjector*). Příklad pro *CarDetailsProjector* je v kódu 8.6. Tento projektor používá třídu *CarCruds* (kód 8.7) a pomocí ní převádí transportní objekty do entit a naopak.

CarFinder (kód 8.8) je třída, která na základě objektu reprezentujícího vyhledávací podmínky a třídy *QueryBuilder* vytvoří dotaz a vrátí požadované výsledky. *QueryBuilder* je třída, která sestavuje samotný SQL dotaz, a v případě použití NoSQL ji lze přizpůsobit odlišnému dotazovacímu jazyku.

SearchParameter je obalující třída (nebo rozhraní) pro třídy *CarFindByModelEq*, *CarFindByLicensePlateEq*, *CarModelFindByCarBrandEq*, *CarModelFindByDayRentalRateLt* a *CarModelFindByDayRentalRateGt*. Například třída *CarFindByLicensePlateEq* (kód 8.9) obsahuje pouze atribut *LicensePlate*.

```
class CarDetailsProjector {

  CarCruds carCruds;

  CarDetails project(CarData data){...}
  void toData(CarData data, CarDetails details){
    carCruds.setModel(data, details.getModel());
    ...
  }
}
```

Zdrojový kód 8.6: CarDetailsProjector

```
class CarCruds {

    CarFinder carFinder;

    DataRef create(CarDetails details) {}
    DataRef modify(CarDetails details) {}
    DataRef delete(DataRef reference) {}
    List<DataRef> find(SearchParameter param){
        carFinder.find(param)
    }
    void setModel(CarData data, DataRef model) {}
    ...
  }
}
```

Zdrojový kód 8.7: CarCruds

```
class CarFinder {

  List<DataRef> find(SearchParameter param) {
    // na zaklade tridy parametru je volana jedna z~
       ↪ metod
  }
  List<DataRef> findByModelEq(SearchParameter param){
    // vytvori QueryBuilder a nastavi parametry
       ↪ dotazu na zaklade param
  }
  List<DataRef> findByLicensePaleteEq(SearchParameter
     ↪  param) {}
  ...
}
```

Zdrojový kód 8.8: CarFinder

Tento atribut se nastaví na požadovanou hodnotu a předá se vyhledávací metodě. Ta na základě jména třídy (pomocí reflexe) zavolá odpovídající vyhledávací metodu a podle atributů nastaví parametry dotazu.

```
1  class CarFindByLicensePlateEq implements
       ↪ SearchParameter {
2
3    String LicensePlate;
4
5    // getters and setters
6  }
```

Zdrojový kód 8.9: CarFindByLicensePlateEq

Z tohoto příkladu je vidět, jak je vybudována struktura tříd datového elementu. Při použití s jinou technologií databáze lze pouze změnit několik drobností a celé napojení bude funkční. Pokud tedy budeme pracovat s databází, pro kterou neexistuje vrstva pracující s rozhraním JPA, bude stačit změnit anotace, pomocí kterých se mapuje do databáze a třídu *EntityManager*. Mnoho NoSQL databází nepodporuje ani jazyk SQL. Pro tyto případy stačí změnit třídy -*Finder* a *QueryBuilder*. Při použití s JPA (v aktuální implementaci) se používá *JPAQueryBuilder* jako konkrétní implementace rozhraní *QueryBuilder*, je tedy možné tuto třídu pro některé entity nahradit jinou třídou generující specifický dotazovací jazyk. Příklad rozhraní *JPAQueryBuilder* je v kódu 8.10, kde objekt třídy *QueryParameter* obsahuje název atributu, operátor a hodnotu parametru. Třída *SortField* obsahuje jméno atributu, podle kterého se má řadit, a směr řazení.

```
1  class JPAQueryBuilder implements QueryBuilder {
2    QueryBuilder addParameter(QueryParameter param) {}
3    QueryBuilder addSortField(SortField sortField) {}
4    // vraci dotaz v~nativnim databszovem jazyku
5    String getDataQueryString() {}
6    // vraci dotaz na pocet polozek
7    String getCountQueryString() {}
8  }
```

Zdrojový kód 8.10: JPAQueryBuilder

Vlastní *EntityManager* nemusí implementovat všechny metody JPA (ani se tak nemusí jmenovat). Může se jednat pouze o jednoduchou třídu, která umožní provádět CRUDS operace s libovolnou entitou. Dokonce je možné mít i zvláštní třídu starající se o persistenci pro každou entitu. Pokud tato třída bude vystavovat stabilní verzi, nepovede její použití ke kombinatorickým efektům.

Závěr

Práce se věnuje teorii normalizovaných systémů a jejich využití v NoSQL databázích. Podle zadání vzniklo několik kapitol zaměřených na různá témata. V první části práce je stručný úvod do NoSQL databází a jejich vlastností, včetně principů CAP a BASE. Další část se věnuje samotné teorii NS a seznamuje čtenáře se všemi požadavky a definicemi. Jedná se o první práci svého druhu v češtině. Jedním z přínosů je i neformální popis normalizovaných systémů. Technologie EJB a existující prostředí založené na EJB, včetně způsobu přístupu k datům je popsán v kapitolách 5 a 6.1. Zbylá část práce je analýza požadavků z formálních definic a existující implementace a studie jejich použití v NoSQL databázích. Jsou zde uvedeny možnosti použití, včetně rozboru jejich vhodnosti a diskuze ohledně výhod a nevýhod daného řešení. V závěru práce je uvedeno několik příkladů použití NoSQL databází a metody při rozhodování, kterou databázi zvolit.

Na začátku práce nebylo jasné, co obsahuje teorie NS a jaké, popřípadě jestli vůbec nějaké požadavky na data klade. Tato teorie byla prostudována a popsána, včetně návazností na existující technologie a programovací paradigmata. Z této analýzy vyplynulo, že požadavky na datovou strukturu nejsou příliš náročné a vyhoví jim většina NoSQL databází. Jejich nasazení je tedy bezproblémové a díky existujícím řešením jako Hibernate OGM nebo Kundera i téměř bez práce. Vhodný výběr databáze pro konkrétní část aplikace může tedy výrazně ovlivnit výkonnost celého systému. Jak se ukázalo, tento výběr může být uskutečněn pouze na základě požadavků samotné aplikace, nikoli podle požadavků NS.

Možnosti využití na ČVUT

V letním semestru akademického roku 2014/2015 proběhne na FIT ČVUT první běh nového předmětu „Normalized Software Systems“, který bude vyučovat jeden ze dvou autorů teorie NS a knihy Normalized systems [15] Jan Verelst. Tato práce by měla posloužit jako jeden z možných studijních ma-

teriálů a také jako první publikace věnující se této teorii, která je napsána v češtině. Pro studium NS poslouží zejména kapitoly 3 a 4, které se věnují existující teorii po formální i neformální stránce. Pozdější kapitoly jsou rozšířením této teorie pro NoSQL databáze.

Možnosti dalšího vývoje

Tato práce se zabývá pouze teoretickým základem, na kterém lze dále stavět. Zde prezentované vlastnosti a požadavky NS lze použít v dalších pracích k vytvoření konkrétní implementace jednotlivých expanderů, které by pracovaly s NoSQL databázemi. Způsobů, jak se postavit k implementaci samotných expanderů, je více a jsou v práci popsány, včetně konkrétních doporučení. Tato práce tedy měla zjistit, zda by bylo možné použít jiné nežli pouze relační databáze a popřípadě jaké. Jednotlivé možnosti byly diskutovány a na základě jejich rešerše je možné přistoupit k samotné implementaci konkrétních expanderů.

O výsledek práce projevili zájem sami autoři teorie NS a proto budou nejdůležitější části přeloženy do anglického jazyka (část práce již přeložena byla) a není vyloučeno rozšíření existující implementace o podporu NoSQL databází.

Literatura

[1] BREWER, Eric. Towards Robust Towards Robust Distributed Systems. In: *Berkeley* [online]. Berkeley, 2000 [cit. 2015-03-10]. Dostupné z: `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`

[2] CAP Twelve Years Later: How the „Rules“ Have Changed. BREWER, Eric. *InfoQ* [online]. 2012 [cit. 2015-03-10]. Dostupné z: `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`

[3] EclipseLink/Examples/JPA/NoSQL. ORACLE. *Eclipse* [online]. 2013 [cit. 2015-03-27]. Dostupné z: https://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL

[4] FOWLER, Martin a SADALAGE. The future is: ~~NoSQL Databases~~ Polyglot Persistence. In: *Http://martinfowler.com/* [online]. 2012 [cit. 2015-03-28]. Dostupné z: `http://martinfowler.com/articles/nosql-intro-original.pdf`

[5] Entity Lifecycle Management. *OpenJPA* [online]. 2013 [cit. 2015-03-26]. Dostupné z: `http://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/jpa_overview_em_lifecycle.html`

[6] Hibernate Object Mapping for NoSQL Datastores. HIGHTOWER, Rick. *InfoQ* [online]. 2011 [cit. 2015-03-24]. Dostupné z: `http://www.infoq.com/news/2011/07/hibernateogm`

[7] HORDĚJČUK, Vojtěch. Voho.cz. *JPA (Java Persistence API)* [online]. 2015 [cit. 2015-03-25]. Dostupné z: `http://voho.cz/wiki/jpa/`

[8] HOUŽVIČKA, Tomáš. *Aplikace NoSQL databází*. Praha, 2012. Dostupné z: `https://is.bivs.cz/th/19233/bivs_m/`. Diplomová práce. Bankovní institut vysoká škola Praha. Vedoucí práce Ing. Michal Valenta Ph.D.

[9] CHURÝ, Lukáš. Aspektově orientované programování v Javě. *Programujte.com* [online]. 2006 [cit. 2015-03-17]. Dostupné z: http://programujte.com/clanek/2006120416-aspektove-orientovane-programovani-v-jave/

[10] Kolařík, Vincenc. *Applying OntoUML for structural definitions of Normalized Systems Expanders.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

[11] KOSEK, Jiří. *Ukládání a vyhledávání XML dat.* 2014 [cit. 2015-03-17]. Dostupné z: http://www.kosek.cz/vyuka/4iz238/slidy/ql.pdf

[12] KROUWEL, Marien. *Towards the agile enterprise: A method to come from a DEMO model to a Normalized System, applied to Government Subsidy Schemes.* Utrecht, 2010. Dostupné z: http://repository.tudelft.nl/assets/uuid:a170e23f-9fee-45fd-b99b-3a85e4d551cf/MScThesisMarienKrouwel-TowardsTheAgileEnterprise.pdf. Master thesis. Department of Software Techonology, Faculty of EEMCS, Delft University of Technology.

[13] Kundera. IMPETUS LABS. *GitHub* [online]. 2014 [cit. 2015-03-27]. Dostupné z: https://github.com/impetus-opensource/Kundera/wiki

[14] LEHMAN, Manny. *Programs, Life cycles, and Laws of Software Evolution.* Proceedings of the IEEE 68(9) [online]. 1980 [cit. 2015-03-12]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.3108&rep=rep1&type=pdf

[15] MANNAERT, Herwig a Jan VERELST. *Normalized systems: Re-creating Information Technology Based on Laws for Software Evolvability.* Paesen: Koppa Digitale Media, 2009. ISBN 978-90-77-160-008.

[16] McIlroy, M.D.: *Mass produced software components.* In: P. Naur, B. Randell (eds.) NATO Conference on Software Engeneering, Garmish, Germany, 7–11 October 1968

[17] *NoSQL: Your Ultimate Guide to the Non-Relational Universe!* [online]. 2009 [cit. 2015-03-10]. Dostupné z: http://nosql-database.org/

[18] ObjectDB: Fast Object Database for Java - with JPA/JDO support. OBJECTDB SOFTWARE. *JPA 2 Annotations* [online]. 2015 [cit. 2015-03-31]. Dostupné z: http://www.objectdb.com/api/java/jpa/annotations

[19] OOST, Arco. NORMALIZED SYSTEMS INSTITUTE. *EuRent: Zdrojový kód ukázkové aplikace.* Antwerp, 2015.

[20] REDHAT. *Hibernate: Hibernate OGM* [online]. 2010 [cit. 2015-03-24]. Dostupné z: `http://hibernate.org/ogm/`

[21] POKORNÝ, Jaroslav a Michal VALENTA. *Databázové systémy*. 1. vyd. Praha: České vysoké učení technické v Praze, 2013, 265 s. ISBN 978-80-01-05212-9.

[22] VALERYIA, Kerol. *NoSQL databázové systémy Redis a Cassandra*. Praha, 2013. BAKALÁŘSKÁ PRÁCE. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky, Katedra informačních technologií. Vedoucí práce Ing. Jarmila Pavlíčková.

[23] SCOFIELD, Ben. *NoSQL - Death to Relational Databases(?)*. [online]. 2010 [cit. 2015-03-17]. Dostupné z: `http://www.slideshare.net/bscofield/nosql-codemash-2010`

[24] SlideShare. WATT, Nicki a Michael HUNGER. *Object Graph mapping: Spring Data Neo4j 3.0* [online]. 2013 [cit. 2015-03-28]. Dostupné z: `http://www.slideshare.net/neo4j/ogm-withsdn30`

[25] Spring.io. *Spring Data Neo4J* [online]. 2015 [cit. 2015-03-28]. Dostupné z: `http://projects.spring.io/spring-data-neo4j/`

[26] TRONÍČEK, Zdeněk. Katedra softwarového inženýrství, Fakulta informačních technologií ČVUT v Praze. *Enterprise Java (BI-EJA)*. Praha, 2012.

[27] VESELÝ, Dominik. *Distribuované ukládání a zpracování velkého množství dat - případové studie*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014. Vedoucí práce Ing. Michal Valenta Ph.D.

Příloha **A**

Seznam použitých zkratek

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

AS Aplikační server

BASE Basic Availability, Soft-state, Eventual consistency

BPEL Business Process Execution Language

CAP Consistency, Availability, Partition tolerance

CRUDS Create, Read, Update, Delete, Search

DTD Document Type Definition

DTO Data transfer object

EJB Enterpsise Java Bean

HTTP Hypertext Transfer Protocol

IO Input-Output (vstup-výstup)

IS Informační systém

JEE Java Enterprise Edition

JPA Java Persistence API

JPQL Java Persistence Query Language

JSON JavaScript Object Notation

JVM Java virtual machine

MVC Model-view-controller

NoSQL not only SQL databases

NS Normalizované systémy

OGM Object Grid Mapping

OOP Objektově orientované programování

OQL Object Query Language

ORM Objektově relační mapování

PNES Předpoklad neomezené evoluce systémů

REST Representational state transfer

SOA Servisně orientovaná architektura

SQL Structured Query Language

SŘDB Systém řízení báze dat

TDV Transparentnost datové verze

XML Extensible Markup Language

Příloha B

Expandované elementy

V této příloze jsou uvedeny struktury jednotlivých elementů pokročilého informačního systému tak, jak jsou popsány v knize [15]. Jde tedy o teoretický popis toho, jak by se daly elementy implementovat v technologiích Java EE, Apache Axis framework, Apache Cocoon a dalších. Konkrétní implementace se však může lišit a příklad pro datový element je uveden v kapitole 8.1.

B.1 Data Element Pattern

- Jednoduchá Java třída s metodami get- a set-.
- Podpůrné úlohy (perzistence, vzdálený přístup), včetně ORM, obstará EJB kontejner.
- EJB také generuje metody pro vyhledávání.
- Dvě DTO třídy (detailní a sumární verze) s defaultním konstruktorem a get- a set- metodami podle modelu JavaBeans.
- Beana je schopna vrátit detailní transportní objekt dané instance nebo kolekci sumárních transportních objektů pro množinu instancí.
- Metody entity beany pokrývají standard CRUDS (Create Retrieve Update Delete Search).

Pattern expansion, neboli generování kódu pro návrhový vzor datových elementů bere následující parametry [15]:

- jméno datového elementu
- kontextové informace (jméno komponenty a package)
- detailní informace o každém datovém fieldu

  - jméno
  - datový typ – pokud je to reference → je uvedeno celé jméno třídy
  - příznaky:
    * info – je-li field přítomen v sumárním DTO
    * find – je-li field dostupný jako klíč pro hledání
    * list – pokud je true → specifikuje možné hodnoty fieldu (enum)

a pomocí těchto parametrů vytvoří (pro element se jménem *Obj*):

- entity bean ($ObjBean$)
  - metody na $ObjBean$ jako:
    * $create(ObjDetails)$
    * $modify(ObjDetails)$
    * $getDetails()$
    * $getInfo()$
- interface:
  - $ObjLocal$
  - $ObjRemote$
- domovské interface:
  - $ObjHomeLocal$
  - $ObjHomeRemote$
- deployment deskriptor
- Dále vytvoří serializovatelné transportní třídy (DTO)
  - $ObjDetails$ – obsahuje všechny položky
  - $ObjInfo$ – obsahuje sumární položky
- EJB-QL – pro vyhledávací metody
- proxy agenta $ObjAgent$ – pro přístup k $ObjBean$ pomocí RMI, který obsahuje metody:
  - $create(ObjDetails)$
  - $modify(ObjDetails)$
  - $delete()$
  - $getDetails()$
  - $getInfoVector()$ – vrátí vektor $ObjInfo$ objektů pro všechny záznamy
  - $getSearchVector()$ – vrátí vektor $ObjInfo$ objektů podle vyhledávacího klíče

B.2 User Connector Element Pattern

Konektor elementy jsou rozděleny na dva druhy. První jsou uživatelské konektor elementy a tím druhým jsou protokolové uživatelské elementy. Element expander bere stejné parametry jako v případě expanze datových elementů a vygeneruje následující strukturu (pro použití s Apache Cocoon frameworkem):

- XML a XSLT dokumenty a Cocoon akce
- ObjDetailer
- ObjEnterer
- ObjRetriever
- konfigurační data do souborů *sitemap.xmap* a *cocoon.xconf*

B.3 Action Element Pattern

Akční elementy využívají OO polymorfismu k zapouzdření akcí a jsou přístupné vzdáleným klientům pomocí RMI. U akčních elementů rozlišujeme mezi dvěma typy vstupních dat:

- Argumenty – data, na kterých jsou akce vykonávány. Jsou povětšinou rozdílné při každém volání akce.
- Parametry – data, která ovlivňují vykonávání jednotlivých akcí. Jsou často vybírány pouze z omezené množiny možností.

Argumenty i parametry jsou zabalené datové elementy. Zatímco argumenty jsou předávány přímo při volání akcí, parametry jsou vloženy do konstruktoru akčních elementů. Implementace může také obsahovat volitelný datový element reprezentující stav.

Pro element se jménem *Act* se vygenerují následující elementy:

- *ActBean* – session bean class
- interface:
  - *ActLocal*
  - *ActRemote*
- domovské interface:
  - *ActHomeLocal*
  - *ActHomeRemote*
- deployment descriptor

- *ActImpl* – obsahuje vzorovou implementaci
- *ActAgent* – zpřístupňuje akci pro vzdálené klienty
- pro argumenty, parametry a případně pro status vytvoří datové elementy:
  - *ActArgum*
  - *ActParam*
  - *ActStatus*

Pomocí expanze tohoto uživatelského konektoru je možné zpřístupnit CRUDS funkcionalitu nad těmito daty.

B.4 Protocol Connector Element Pattern

Pro tyto konektory používají [15] Apache Axis framework.

Pattern expander vytvoří pro data element s názvem *Obj* a to samé i pro vzdálený přístup *Net* následující strukturu:

- *ObjProxyBean* – bezstavová EJB session bean s metodami:
  - *create*
  - *modify*
  - *delete*
  - *getInfo*
  - *getDetails*
  - *getInfoVector*
  - *getSearchVector*
- interface:
  - *ObjProxyLocal*
  - *ObjProxyRemote*
- domovské interface:
  - *ObjProxyHomeLocal*
  - *ObjProxyHomeRemote*
- deployment descriptor a nějaká další konfigurační data
- pro binární nebo proprietální protokol vytvoří:
  - *ObjConnectBean* – EJB session bean

  - interface:
    * *ObjConnectLocal*
    * *ObjConnectRemote*
  - domovské interface:
    * *ObjConnectHomeLocal*
    * *ObjConnectHomeRemote*
  - depolyment descriptor
  - *ObjConnectImpl* – vzorovou implementaci specifického protokolu
  - v případě síťového komunikačního protokolu:
    * *NetConnectBean* – session beana
    * interface, deployment descriptor
    * *NetConnectImpl* – vzorová implementace

B.5 Workflow Element Pattern

Předpokládejme workflow element se jménem *Flow*, který pracuje nad datovým elementem s názvem *Target*. V tomto případě pattern expander vygeneruje následující strukturu:

- *FlowBean* – session bean s kódem přechodů mezi jednotlivými stavy
- interface:
  - *FlowLocal*
  - *FlowRemote*
- domovské interface:
  - *FlowHomeLocal*
  - *FlowHomeRemote*
- depolyment descriptor

B.6 Trigger Element Pattern

Uvažujeme-li trigger element s názvem *Srv*, pattern expander pro něho vytvoří Jonas service a zadefinuje ji v konfiguračním souboru. Tato servisa bude v pravidelných intervalech kontrolovat datový element *ServiceEngine* a provádět akce specifikované ve workflow elementu *Flow*.

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS